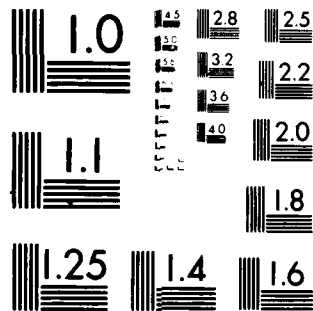END
DATE
FILMED
1-82
DTIC

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

NRL Report 8514

A PRF Sorter Based on List Manipulation Techniques

J. O. COLEMAN

Radar Analysis Branch
Radar Division

November 20, 1981

NAVAL RESEARCH LABORATORY
Washington, D.C.

81 12 09 021

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
NRL Report 8514	AD-A108365	

4. TITLE *(and Subtitle)*	5. TYPE OF REPORT & PERIOD COVERED
A PRF SORTER BASED ON LIST MANIPULATION TECHNIQUES	Interim report on a continuing NRL problem
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(*s*)	8. CONTRACT OR GRANT NUMBER(*s*)
J. O. Coleman	

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Naval Research Laboratory Washington, DC 20375	62712N; XF12-151-104; NRL Problem 53-0612-00-1

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Department of the Navy	November 20, 1981
Naval Electronics Systems Command	13. NUMBER OF PAGES
Washington, DC 20376	61

14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)	15. SECURITY CLASS. *(of this report)*
	UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

PRF sorting
Pulse sorting
De-interleaving
Sorting

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes a program that uses list processing techniques to sort pulses into sequences uniformly spaced in time, a process referred to as PRF sorting. The detailed discussion of the program's operation is based on a LISP implementation that functions on a demonstration level. A listing of a PASCAL version capable of operation on real data is also given. The unique feature of the algorithm used here is the ability to postpone the decision to assign a pulse to a particular existing pulse train. This is achieved by duplicating the existing

(Continued)

DD FORM 1473   EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73

S/N 0102-014-6601

i

20. ABSTRACT *(Continued)*

pulse train and assigning the pulse to one of the two resulting pulse trains. Both pulse trains are then carried in the system, with equal status, until eventually a decision is made as to which is correct. The list structures used, native to LISP, prevent the duplication of pulse trains from resulting in unreasonable storage requirements.

CONTENTS

Accession For

NTIS GRA&I   X
DTIC TAB   ☐
Unannounced   ☐
Justification

By
Distribution/
Availability Codes
         Avail and/or
Dist   Special

A

A PRF SORTER BASED ON LIST MANIPULATION TECHNIQUES

INTRODUCTION

This report describes a program which takes a temporally ordered list of pulses and sorts the pulses into groups, each of which is characterized by a constant pulse repetition frequency (PRF). Such algorithms are generally known as "PRF sorters" or "pulse de-interleavers." Our design requirements were twofold. First, a "reasonably" capable algorithm was needed to sort experimental data previously stored on magnetic tape. Occasional placement of a pulse in an incorrect group was considered tolerable. While real-time operation was not necessary, the program was required to sort large collections of input data (millions of pulses) for a reasonable cost on NRL's Advanced Scientific Computer (ASC). Second, as this algorithm and program design were part of a research project directed primarily at other ends (radar/ESM integration), the design itself needed to be completed as quickly and inexpensively as possible. Because the cost of development would likely dominate the life-cycle cost of the software, this second requirement was the more important factor.

The choice of a programming language was restricted by the availability of languages on the ASC (where the input data files were available) to FORTRAN, RATFOR (a FORTRAN preprocessor adding structured control constructs to FORTRAN), or PASCAL. Because the Sorter would involve extensive manipulation of lists of pulses, FORTRAN and RATFOR seemed inappropriate. PASCAL was therefore the language of choice. (The input routines for the PASCAL version of the PRF Sorter were actually written in RATFOR to be compatible with ASC file handling software.) Unfortunately, the programming environment on the ASC (a poor editor, no interactive debugging, and a clumsy job control language) made the prospect of undertaking the initial development of an algorithm on the ASC rather unappealing. The decision was therefore made to do the development of the algorithm in LISP on a different computer system, the MIT MACSYMA Consortium machine,* thereby gaining time through the use of superior editing facilities and a naturally interactive language with excellent debugging features. Since LISP is first and foremost a list processing language, it fit the problem well. (While I do not maintain that the algorithm presented here is sufficiently developed for use in a piece of physical equipment operating in real time, the recent demonstration of a microprocessor which executes LISP primitives directly [1,2] implies that a LISP-based approach to the sorting problem may be more practical than its use as a development tool might imply.) Once an algorithm had been successfully developed and debugged in LISP it could be translated to PASCAL. This turned out to be a good strategy since a number of approaches to the sorting problem were easily explored in LISP before settling on one for the final implementation.

As it turned out, the LISP version is both much shorter and easier to understand than the PASCAL translation. Consequently, the LISP version is presented here. The next section gives an overview of the sorting algorithm in general terms. A section on MACLISP, the dialect of LISP used, follows. The Sorter is then discussed in detail, with the source code interspersed with the text. The report concludes with a description of the performance of the PASCAL translation and a short summary.

OVERVIEW OF THE SORTING ALGORITHM

The input to the Sorter is a stream of pulses ordered by time of arrival. The Sorter output consists of groups of pulses. Each group, also referred to as a pulse train or simply "train," contains

1

member pulses that could have been drawn from a source with a uniform PRF. For example, the times of arrival of the pulses within a group might form an arithmetic sequence like

$$(10\ 13\ 16\ 19\ 22),$$

or the times might form an arithmetic sequence with some elements missing, like

$$(10\ 16\ 19\ 25\ 28).$$

The Sorter operates sequentially, taking in a pulse at a time and putting out a pulse train whenever it becomes certain a train is complete. Although the trains come out approximately in chronological order, there is no guarantee that a train finishing earlier will be output before a train finishing later.

The sequence of operations in the Sorter is as follows:

1. Get the next pulse.

2. Add the pulse to any trains for which the pulse is a "perfect fit." A perfect fit would be a situation in which the interval between the time of arrival of the new pulse and the time of arrival of the last pulse already in the train matched exactly the interval between the times of arrival of the last two pulses in the train. If the intervals do not match quite closely enough, it may be considered "fits OK" rather than a perfect fit.

3. If there were no perfect fits in the previous step, split (duplicate) each train for which the pulse fits OK into two trains, add the pulse to one only, and start a new train consisting only of the new pulse. If there are no fits at all, start a new train with the pulse.

4. Output any trains that have become "old" (i.e., that have no chance of ever being updated again) after resolving the conflicts (pulses in common) with other trains.

5. Repeat all of the above until no more pulses are available.

6. Finally, resolve the conflicts among the remaining trains and output them.

An important feature of the Sorter is that whenever it is unclear whether or not a particular train should have the pulse added both options can be temporarily accepted, thus postponing the real decision until more information is available. This introduces two potential difficulties. The first involves storage management. Since trains are frequently split into two trains, this strategy has the potential for requiring an enormous amount of storage. This problem is solved nicely by the use of the list handling facilities of LISP (the core of these facilities was duplicated in the PASCAL translation so as to provide the same benefits), and as a side benefit, there are no arbitrary limits on the allowed numbers of particular items in the system, e.g., the number of trains or the number of pulses in a train. The only limit on storage is a limit on the combined storage used for trains, lists of trains, and pulses. (While this isn't strictly true in the PASCAL translation, a similar statement, almost as strong, would apply there as well.) The mechanism providing these benefits will be mentioned briefly in the detailed description of the Sorter. For a more thorough description, see chapter 9 of Ref. 3.

The second potential difficulty introduced by train splitting is in the conflict resolution process. Since trains, once split, have equal status, there will eventually be a time (probably after both have been updated with additional pulses) when a decision will have to be made as to which is the "better" train, that is, the train most likely to be correct. While no definitive solution to this problem has been reached, the simple measures of train quality used here seem to result in satisfactory sorting of our experimental data. Better measures of train quality might be required for more demanding applications.

2

SOURCES OF MACLISP INFORMATION

The discussion of the Sorter in the following section is in sufficiently general terms that a reader with little or no understanding of LISP should be able to follow most of it. However, since the discussion of the Sorter is based directly on the MACLISP source code, the reader does need some familiarity with the language to understand the correspondence between the discussion and the code itself. It is relatively simple to gain a working knowledge of LISP, and there are several good texts available. My favorite text is Winston and Horn [3], which has the advantage of being based on the MACLISP dialect. For the purposes of this report all that is really necessary is an understanding of LISP equivalent to the material covered in the first three chapters of that book plus parts of chapters 6 and 9. This is true because the Sorter was written in a fairly narrow subset of LISP so that it could be straight-forwardly translated to PASCAL. Other source books for an introduction to LISP are Siklossy [4] and Winston [5].

There are two MACLISP forms (functions) used in the Sorter, LET and DO, which are not discussed in the texts because they are not yet standardized across all LISP dialects. They will therefore be described briefly here. The reader with no familiarity with LISP whatever would do well to skip directly to the discussion of the Sorter in the next section.

A fairly intuitive definition of the macro LET is [6]

```
(LET ((A <e1>) (B <e2>) ... (C <en>))
     <computate>)
```

macro-expands into

```
((LAMBDA (A B ... C)  <computate> )
        <e1> <e2> ... <en>)
```

The DO special form (this discussion is edited from Moon [7]) provides a generalized "do loop" facility, with an arbitrary number of "index variables" whose values are saved when the DO is entered and restored when it is left; i.e., they are bound by the DO. The index variables are used in the iteration performed by DO. At the beginning they are initialized to specified values, and then at the end of each trip around the loop the values of the index variables are changed according to specified rules. DO allows the programmer to specify a predicate which determines when the iteration will terminate. The value to be returned as the result of the form may optionally be specified.

DO looks like

```
(DO ((var init repeat)...)
    (end-test exit-form...)
    body...)
```

The first item in the form is a list of zero or more index variable specifiers. Each index variable specifier is a list of the name of a variable VAR, an initial value INIT which defaults to NIL if it is omitted, and a repeat value REPEAT. If REPEAT is omitted, VAR is not changed between loops.

All assignment to the index variables is done in parallel. (In the Sorter care was taken not to depend on this feature.) At the beginning of the first iteration, all the INITs are evaluated, then the VARs are saved and then SETQed to the values of the INITs. To put it another way, the VARs are lambda-bound to the values of the INITs; the INITs are evaluated before the VARs are bound. At the beginning of each succeeding iteration those VARs that have REPEATs get SETQed to the values of their respective REPEATs. All the REPEATs are evaluated before any of the VARs are changed.

The second element of the DO form is a list of an end-testing predicate END-TEST and zero or more forms, the EXIT-FORMs. At the beginning of each iteration, after processing of the REPEATs,

3

the END-TEST is evaluated. If the result is NIL, execution proceeds with the BODY of the DO. If the result is not NIL, the EXIT-FORMs are evaluated from left to right and then DO returns. The value of the DO is the value of the last EXIT-FORM, or it is NIL if there were no EXIT-FORMs. The second element of the DO-form resembles a COND clause.

The remainder of the DO-form constitutes a PROG-body. When the end of the BODY is reached, the next iteration of the DO begins.

DETAILS OF THE PRF SORTER

In this section the LISP code making up the Sorter is described in detail. The Sorter is made up of many functions, each of which is discussed individually below. As an aid to remembering the relationships between the various functions, Table 1 shows "who calls whom." The format is similar to an outline. Each function directly calls those functions shown below it indented one more level. For example, DECIDE directly calls REMOVE-PULSES and BEST. It is important to realize, however, that Table 1 shows only the "static" structure of the program. The order in which functions are shown in the table corresponds to the order in which they appear in the program code, rather than the chronological order in which they are called.

The function SORT is the equivalent of a "main" program. It accepts a single argument, PULSES, presumed to be an ordered list of pulses with the earliest pulses at the beginning of the list, and returns a list of pulse trains. Each pulse train in turn contains those pulses which the program has decided belong together. An example should help make this clear. If, after loading the entire program into LISP from a file, the user were to type

(sort '(0 1 5 6 10 11 15 16 20))

the LISP system would respond by typing out

((20 15 10 5 0) (16 11 6 1))

which is the obvious way to sort the sequence given. Notice that the user's input is shown somewhat indented from the rest of the code. This is not a characteristic of the LISP system (as it would be, for example, on most systems using the APL language) but was added to clarify the presentation. This convention will be used for the remainder of this report.

A few comments on data structures are in order before the code itself is discussed. Many of the functions making up the Sorter deal with lists of pulse trains. Knowledge of the internal structure of a train is restricted, however, to a small handful of functions for testing and manipulating trains. This ensures that if the need should arise to change the representation, perhaps to include more features, all would not be lost. As it turned out, the simplest possible representation of a train was kept throughout the development. A train is simply a chronologically ordered list of pulses, with the most recent pulses (those with the chronologically latest times of arrival) at the head of the list.

The detailed knowledge of the representation of a pulse is confined to a single function. Again the representation used in this development was the simplest possible, representing a pulse by its time of arrival. A somewhat more complex representation was used in the PASCAL translation, as other (than time) data pertaining to the pulse needed to be kept for later use.

In the material that follows, the actual LISP code is given interspersed with the corresponding discussion in the text. The code is suitable for feeding directly into MACLISP in the order given, and it

4

Table 1. The Function-Calling Structure of the Sorter

```
SORT
    GROUP
        GROUPS
            GET-PULSE
            PUT-PULSE
            DECIDE                                      †
                REMOVE-PULSES                           †
                    TRAIN-MINUS
                BEST
                    BETTER-OF
            UPDATE
                UPDATE-PERFECT
                    INIT-TRAIN
                    PERFECT-FIT
                        NR-PULSES                       *
                        TOA                             *
                        LATEST-PULSE                    *
                        NEXT-LATEST-PULSE               *
                        ADD-PULSE
                INIT-TRAIN
                UPDATE-OK
                    INIT-TRAIN
                    FITS-OK
                        ADD-PULSE
                OUTPUT-AGED
                    OLD
                    QUAL-TRAIN
                    TRAIN-MINUS
                    REMOVE-PULSES                       †
                    DECIDE                              †
```

* Indicates a "utility" function.
  Only the first use shown.

† Indicates a function called elsewhere also.
  Functions it calls shown with first use only.

has been executed directly from the computer-stored manuscript. Therefore, unless errors are introduced in typesetting, the code should be accurate. An attempt was made to describe the general operation of the Sorter so that it can be understood by readers who are not familiar with LISP. Such readers should probably skip not only the LISP code but the footnotes in this section as well, as they were intended for those with a desire to understand the LISP at a very detailed level.

Here is the definition of (i.e., code for) SORT, the "top-level" function:

```
(defun sort (pulses)
    (let ((old-groups nil)
          (trains nil)
          (max-pri 10))
        (do ((this-group (group) (group))
             (result nil (cons this-group result)))
            ((not this-group) result)))))
```

5

The SORT function itself does two things. It initializes some "global" variables used in subordinate functions, and it repeatedly calls on the function GROUP. GROUP returns the "next" sorted pulse train, where "next" means the next to be determined by the Sorter, which may or may not be the next in the sense of pulse times of arrival. If there are no more pulse trains (e.g., if the end of an input file has been reached), GROUP returns NIL, equivalent in LISP to the empty list. SORT simply takes the pulse trains returned by GROUP and accumulates them into a list to be returned upon detecting a NIL result from GROUP. SORT is really just a convenient interface for the development of the the Sorter, having nothing to do with the sorting process itself. In a practical implementation in which the Sorter was embedded in a larger system, GROUP would serve as the interface to the Sorter:

```
(defun group ()                    ; free: old-groups
       (cond (old-groups
              (let ((grp (car old-groups)))
                   (setq old-groups (cdr old-groups))
                   grp))
             ((setq old-groups (groups)) (group))))
```

The function GROUP simply maintains a buffer of sorted pulse trains, OLD-GROUPS, feeding pulse trains one at a time to the calling routine and calling on the function GROUPS to refill the buffer when it becomes empty (NIL). Note that OLD-GROUPS serves here as a "static" variable, that is, one whose value is preserved across successive calls to GROUP. This effect is achieved by having OLD-GROUPS be a "nonlocal" variable or, in LISP parlance, be free with respect to GROUP. OLD-GROUPS was initialized (bound) to NIL in SORT.

A comment is in order here on recursion. While recursion in many contexts can be quite efficient in MACLISP (and frequently the most transparent way to express an algorithm), it is fairly inefficient in the locally available implementation of PASCAL. Since the intent in this development was to translate eventually to PASCAL, recursion was generally avoided in the Sorter. An exception was made for the function GROUP (which calls itself in the last line), because it is called so few times (relative to most routines in the Sorter) that the efficiency penalty is negligible. A side effect of the general avoidance of recursion in the Sorter was to make much of the code longer and a bit more obscure than it might otherwise have been. (Be warned, therefore, that nothing in the Sorter should be assumed to represent generally good or desirable LISP programming style.)

The function GROUPS is where the real management of the sorting process takes place:

```
(defun groups ()                     ; free: trains
       (let ((out-trains nil))
            (do ((pulse (get-pulse) (get-pulse)))
                ((or out-trains (not pulse))
                 (cond (out-trains
                        (put-pulse pulse))
                       (trains
                        (setq out-trains (decide trains))
                        (setq trains nil))))
                (setq trains (update pulse trains))
                (setq out-trains (output-aged pulse)))
            out-trains))
```

GROUPS divides the sorting process into two parts. The first uses pulses as they are obtained to update a list of hypothetically possible pulse trains being maintained in TRAINS. This is done using the function UPDATE. The other major portion of the sorting process, performed by OUTPUT-AGED and DECIDE, removes completed pulse trains from TRAINS for output. As will become evident when

these functions are discussed, this is the more difficult task. The functions GET-PULSE and PUT-PULSE are used to manage the input stream of pulses. The operation of GROUPS will now be described.

After initializing the variable OUT-TRAINS to NIL, the function GROUPS enters a loop which does the following: The next available pulse is obtained through a call to the function GET-PULSE and stored in the variable PULSE. A test is made that will exit the loop if either PULSE is NIL, indicating no more input pulses exist, or OUT-TRAINS is non-NIL, indicating that some pulse trains are ready for output. Assuming for the moment that the test fails, the function UPDATE is called to use PULSE to update the value of TRAINS. TRAINS is free with respect to GROUPS, and is the list of possible not-yet-completed pulse trains. Once TRAINS has been updated, OUTPUT-AGED is called to transfer to OUT-TRAINS any pulse trains in TRAINS which are complete. Of course, OUT-TRAINS has the side effect of changing TRAINS accordingly. This loop is repeated until the exit test is passed. At this time, if OUT-TRAINS is non-NIL, indicating that OUTPUT-AGED has indeed taken some action, PUT-PULSE is called to return the pulse just acquired by GET-PULSE to the input stream for later use. Of course the pulse is in reality buffered, but this is relevant only to GET-PULSE and PUT-PULSE. GROUPS is then exited with OUT-TRAINS returned as its value. If OUT-TRAINS was NIL following the exit test, the test must have passed due to a failure of GET-PULSE to return a pulse, implying that the sort must be brought to an end. In this event DECIDE is called to make the final choices as to which pulse trains in TRAINS are valid, putting them in OUT-TRAINS to be returned as the value of GROUPS.

The version of GET-PULSE shown here returns the first element (in LISP called the CAR) of the list PULSES while removing that same element from PULSES using CDR (the complement of CAR) to obtain all but the first element of its argument:

```
(defun get-pulse ()
        (let ((pulse (car pulses)))
                (setq pulses (cdr pulses))
                pulse))
```

Recall that PULSES was the argument to the function SORT. This is sufficient for algorithm development, but like SORT itself, would not exist in the same form in a system operating on real data.

The function PUT-PULSE simply CONSes (mnemonic: CONS means construct) its argument back onto the front of PULSES:

```
(defun put-pulse (pulse)
        (setq pulses (cons pulse pulses)))
```

The function UPDATE uses its argument PULSE to construct a new list of possible pulse trains from its argument TRAINS, returning the newly constructed list as its value:

```
(defun update (pulse trains)
        (cond ((update-perfect pulse trains))
           ((cons (init-train pulse)
                   (update-ok pulse trains)))))
```

The task is divided into two parts. UPDATE-PERFECT is first called to add PULSE to any trains in TRAINS where the new pulse is an "exact" fit in some sense. UPDATE-PERFECT doesn't actually change TRAINS but returns the newly updated list as its value, which is then immediately returned as the value of UPDATE. If PULSE is not an exact fit for any of the trains in TRAINS, UPDATE-PERFECT returns NIL, and UPDATE responds by calling UPDATE-OK. UPDATE-OK is similar to

7

UPDATE-PERFECT with two exceptions. First, a less perfect fit of PULSE to a train is required. Second, when a fit is obtained, the associated train is "split," that is, the train becomes two trains which are identical except that one contains the new PULSE and one does not. This, in effect, expresses the notion that PULSE may belong with that train, but it may not; therefore both possibilities are retained. The conflict will eventually have to be resolved by OUTPUT-AGED or DECIDE. If UPDATE-OK finds no fit at all, it returns TRAINS unchanged. Since it is certainly true that if PULSE fit none of the trains perfectly one possibility is that it is the begining of a new train, UPDATE adds one more train to the list returned by UPDATE-OK before returning the list of trains as its value. That train is created by INIT-TRAIN and consists of the single pulse PULSE.

This train-splitting action is really the heart of the Sorter and is where it differs most from other pulse sorting algorithms. It is an attempt to prevent a wrong decision on a particular pulse from becoming permanent. The obvious penalty is that the number of trains being maintained at any one time tends to be somewhat greater than the actual number of correct trains in existence at that time. This leads to an obvious speed penalty. In addition, it might appear to those readers not familiar with LISP that there would be both a heavy storage penalty and a further speed penalty from the duplication of the existing portion of a train when the train is split. However, due to the way lists are implemented in LISP (a similar implementation was used in the PASCAL translation), the existing train is not copied at all but is represented in the new train by a pointer to the old train. Similarly, even though a pulse may be part of several trains at any one time in the sorting process, the representation of the pulses in the trains uses pointers. Thus, only one copy of the data representing the pulse actually exists. While these data consist only of, time of arrival in this MACLISP version, in general many parameters could be associated with each pulse. For a more thorough description of the representation of lists (and other objects) in LISP, see Ref. 3, chapter 9.

The function UPDATE-PERFECT checks first for the prior existence of pulse trains:

```
(defun update-perfect (pulse trains)
      (cond ((not trains)
             (list (init-train pulse)))
            ((do ((this-train (car trains)
                              (car rem-trains))
                  (rem-trains (cdr trains) (cdr rem-trains))
                  (any-perfect nil (or any-perfect fitted))
                  (up-trains nil
                             (cons (cond (fitted)
                                         (this-train))
                                   up-trains)))
                 (fitted))
                ((not this-train)
                 (cond (any-perfect up-trains) (nil)))
                (setq fitted
                      (perfect-fit pulse this-train)))))))
```

If there are no existing trains, indicated by a value of NIL for TRAINS, a new pulse train is constructed of the single pulse PULSE, and the new train is made the only element of a list and returned as the value of the function. In the more common situation, in which TRAINS is not an empty list, UPDATE-PERFECT enters a loop which calls PERFECT-FIT on each train in TRAINS in succession to test for the possibility that PULSE belongs with that train. If PERFECT-FIT returns NIL, the train is put unchanged into the output list UP-TRAINS. If a fit is recognized, PERFECT-FIT will return the updated train, which is then substituted for the original train in UP-TRAINS. The variable ANY-PERFECT is used to detect the situation in which PERFECT-FIT returns NIL for all of the trains in TRAINS. In this case UPDATE-PERFECT returns NIL, otherwise it returns UPDATE-TRAINS.

The PERFECT-FIT function shown here is suitable for algorithm development on simple pulse sequences (i.e., with integer times):

```
(defun perfect-fit (pulse train)
       (cond ((> 2 (nr-pulses train))
              nil)
             ((eq (toa pulse)
                  (- (* 2 (toa (latest-pulse train)))
                     (toa (next-latest-pulse train))))
              (add-pulse pulse train))))
```

This function indicates a fit if and only if the train contains at least two pulses and the difference between the times of the last two pulses is exactly equal to the difference between the time of the last pulse and the new pulse under consideration, PULSE. This is obviously not suitable for use with real data where times are measured with finite accuracy, and therefore this function's operation was changed somewhat in the PASCAL translation. The details are described in comments in the PASCAL listing of the function.

The only function in the Sorter embodying any knowledge of the (rather trivial) representation of a pulse is TOA, which returns a pulse's time of arrival:

```
(defun toa (pulse)
       pulse)
```

Here are five short functions which exist only to isolate the functions that use them from the details of the representation of pulse trains:*

```
(defun init-train (pulse)
       (list pulse))
```

```
(defun add-pulse (pulse train)
       (cons pulse train))
```

```
(defun nr-pulses (train)
       (length train))
```

```
(defun next-latest-pulse (train)
       (car (cdr train)))
```

```
(defun latest-pulse (train)
       (car train))
```

INIT-TRAIN simply creates a list containing only the single PULSE given as its argument. ADD-PULSE uses CONS (a LISP primitive) to create a new list consisting of PULSE followed by the elements of TRAIN. NR-PULSES returns the number of pulses in the train which is its argument. LATEST-PULSE returns the pulse in its argument with the latest time of arrival. Similarly, NEXT-LATEST-PULSE returns the pulse in its argument with the second latest time of arrival. Besides these five, the only other function in the Sorter which embodies any knowledge of the representation of a train is TRAIN-MINUS (to be discussed later).

---

*Clearly these are so trivial (as is TOA) that simple macro definitions would suffice. This was in fact done in the PASCAL translation. The functional form was kept here to simplify the presentation.

The operation of UPDATE-OK is similar to the operation of UPDATE-PERFECT (as described earlier):

```
(defun update-ok (pulse trains)
       (cond ((not trains)
              (list (init-train pulse)))
             ((do ((this-train (car trains)
                               (car rem-trains))
                   (rem-trains (cdr trains) (cdr rem-trains))
                   (up-trains trains
                              (cond (fitted (cons fitted
                                                  up-trains))
                                    (up-trains))))
                  (fitted))
                 ((not this-train) up-trains)
                 (setq fitted (fits-ok pulse this-train)))))))
```

The differences are: First, FITS-OK is used as the test rather than PERFECT-FIT. In addition, since all of the original trains must be returned regardless of whether any fit is detected, UP-TRAINS is initialized to TRAINS at the beginning of the loop rather than to NIL. From then on, UP-TRAINS is only modified when necessary (as determined by FITS-OK) to add the new train which results when a train is split. UP-TRAINS is always returned as the value of UPDATE-OK when the loop is exited.

In this version of FITS-OK a fit is detected only for the case of a train consisting of a single pulse, since any second pulse certainly suffices to create a hypothetically possible train:

```
(defun fits-ok (pulse train)
       (cond ((greaterp 2 (nr-pulses train))
              (add-pulse pulse train))))
```

If a fit is detected, the value returned is the train with the new pulse added. For operation on real data it would be appropriate to extend this criterion for detecting a fit. This was done in the PASCAL translation and is described in comments in the PASCAL listing of the function.

The function OUTPUT-AGED (called from GROUPS) is responsible for finding and removing trains from TRAINS which have become old. A train is old when there is no possibility that it will be updated further. (See the function OLD, described later.) The basic assumption is that an old train is complete and should be output. However, since a given pulse may have appeared in more than one train in TRAINS, and since designating a train for output implies that the proper train has been found for those pulses, OUTPUT-AGED must insure that any pulses in trains designated for output do not appear in TRAINS, and vice-versa. This turns out to be the major part of the work required of OUTPUT-AGED:

```
(defun output-aged (pulse)              ; free: trains
       (let ((time (toa pulse))
             (out-trains nil))

             ;; Split off old trains:
             (do ((this (car trains) (car rem))
                  (rem (cdr trains) (cdr rem)))
                 ((not this))
                 (cond ((old this)
                        (setq trains (delq this trains))
                        (setq out-trains
                              (cons (append this nil) out-trains)))))
```

10

```
;; Conflict resolution between TRAINS and OUT-TRAINS:
;; thin out...
(do ((this-old (car out-trains) (car rem))
     (rem (cdr out-trains) (cdr rem))
     (thin-old nil (cond (this-old
                                (cons this-old
                                        thin-old))
                         (thin-old)))))
     ((not this-old) (setq out-trains thin-old))
     ;; ...old trains:
     (do ((this-train (car trains)
                        (car rem-trains))
          (rem-trains (cdr trains)
                        (cdr rem-trains)))
         ((or (not this-train) (not this-old)))
         (cond ((not (> (qual-train this-old)
                        (qual-train this-train)))
                (setq this-old
                        (train-minus this-old
                                        this-train)))))
     (setq trains
          ;;...current trains
          (remove-pulses this-old
                            trains)))

     ;; Conflict resolution within TRAINS:
     (decide out-trains)))
```

The detailed operation of OUTPUT-AGED can be described as a sequence of four operations:

Initialization. The variable TIME, used later by the function OLD, is set to the time of arrival of PULSE, the argument to OUTPUT-AGED. The variable OUT-TRAINS, used to accumulate trains for output, is initialized to the empty list.

Splitting off old trains. A loop calls the function OLD on each train in TRAINS. Old trains, indicated by a non-NIL result from OLD, are deleted from TRAINS and added to OUT-TRAINS.*

Conflict resolution between TRAINS and OUT-TRAINS. It is here that pulses are restricted to appearing in either TRAINS or OUT-TRAINS but not both. The code to accomplish this makes up the bulk of the function and will be discussed in detail later.

Conflict resolution within OUT-TRAINS. Here pulses appearing in OUT-TRAINS are restricted to appearing in only one train. This is accomplished using the function DECIDE,† whose result is returned as the result of OUTPUT-AGED.

---

*Since, due to the way the updating process operates, trains in TRAINS may in general have common subtrains, APPEND is used to make sure that it is actually a copy of each old train that is added to OUT-TRAINS. This will prevent certain later pulse deletions from having the undesirable side effects of also deleting pulses from trains sharing the same subtrain. The places where this is important will be noted when that part of the code is discussed.

†The fact that OUT-TRAINS at this poin⁺ contains only copies is not important, because DECIDE takes its own precautions against deletion side effects.

11

Conflict resolution between TRAINS and OUT-TRAINS proceeds as follows: A nested loop arrangement is used to compare the relative quality, as determined by the function QUAL-TRAIN, of each train in OUT-TRAINS with each train in TRAINS. On each such comparison, if the train from TRAINS is determined to be the superior, the function TRAIN-MINUS is used to remove any pulses the two trains have in common from the train in OUT-TRAINS.* Of course, if all of the pulses are removed from a train in this fashion, the train itself is removed. This assures the eventual removal of the superfluous train created in the train-splitting performed by UPDATE-OK. The key to this process is the nature of the function QUAL-TRAIN. While a simple function was used in this development, a good deal of power could be added to the Sorter by using a better train-rating strategy in this function.

The conflict-resolution process must then remove any pulses which are left in OUT-TRAINS from TRAINS.† This is done with the function REMOVE-PULSES.‡ After this step the set of pulses in OUT-TRAINS and the set of pulses in TRAINS are mutually exclusive.

The function OLD determines whether a particular train has become old by comparing TIME, set in OUTPUT-AGED to the time of the last pulse used, to a threshold computed from the train:

```
(defun old (train)              ; non-local: time, max-pri
        (cond ((> time
                  (cond ((eq 1 (nr-pulses train))
                         (+ max-pri (toa (latest-pulse train))))
                        ((- (* 2 (toa (latest-pulse train)))
                            (toa (next-latest-pulse train)))))))))
```

For trains of two or more pulses this threshold is simply the expected time of the next pulse in the train calculated by extrapolating from the two latest pulses. (A small addition to this threshold would be necessary for operation with real data consisting of imperfect measurements.) For trains of a single pulse the threshold is set at the pulse time plus MAX-PRI, a constant set in SORT.

For development purposes, QUAL-TRAIN simply looked at the number of pulses in the train and assigned a quality factor equal to one for trains of one pulse, two for trains of two pulses, and three for all other trains:

```
(defun qual-train (train)
        (let ((l (nr-pulses train)))
             (cond ((< l 4) l)
                   (3))))
```

Other factors which could be used in a more sophisticated version of QUAL-TRAIN include the total time period spanned by the train and the number of missing pulses in the train.§

TRAIN-MINUS and REMOVE-PULSES are really utility functions whose operation is not related to the sorting process; therefore, only a brief description of each is given. TRAIN-MINUS returns its first argument after removing any pulses the first and second arguments may have in common:**

---

*It is important that TRAIN-MINUS is operating on a copy of the original old train, rather than on the original train itself, since TRAIN-MINUS (as will be seen later) uses DELQ to remove pulses from its first argument.
†This is actually accomplished within the outermost of the nested loops mentioned earlier.
‡Again, the fact that the trains in OUT-TRAINS are actually copies of the trains that appeared in TRAINS is important, because REMOVE-PULSES uses TRAIN-MINUS, which uses DELQ.
§Recall that OUTPUT-AGED occasionally removes pulses from trains. In addition, a more sophisticated version of FITS-OK might allow UPDATE-OK to update a train when there is a missing pulse.
**TRAIN-MINUS obviously must incorporate knowledge of the structure of a train.

12

```
(defun train-minus (a b)
       (cond((not a) nil )
            ((do ((this-b (car b) (car rem-b))
                  (rem-b (cdr b) (cdr rem-b))
                  (up-a a (delq this-b up-a)))
                 ((not this-b) up-a)))))
```

In REMOVE-PULSES the variables have been named to suggest its application in the function
DECIDE (to be discussed later). REMOVE-PULSES returns its second argument, a list of trains, after
removing all occurrences of pulses common to its first argument, a train.*

```
(defun remove-pulses (best-train trains)
       (do ((this-train (car trains) (car rem-trains))
            (rem-trains (cdr trains) (cdr rem-trains))
            (up-trains ()
                          (let ((stripped
                                  (train-minus this-train
                                                       best-train)))
                                (cond ((and stripped
                                            (not
                                              (memq stripped
                                                      up-trains)))
                                       (cons stripped
                                             up-trains))
                                      (up-trains)))))
           ((not this-train) up-trains)))
```

The function DECIDE takes as its argument a list of trains, and it returns a list of trains contain-
ing mutually exclusive sets of pulses; that is, it resolves the conflicts among the trains. It does this by
first initializing REM-TRAINS to the train list and then entering a loop which repeatedly does the fol-
lowing:

1. It finds the "best" train in REM-TRAINS, using the function BEST, and assigns it to the
    variable BEST-TRAIN†

2. Using REMOVE-PULSES, it removes from all other trains any pulses which they have in
    common with BEST-TRAIN. This has the beneficial side effect of removing BEST-
    TRAIN from REM-TRAINS.

3. It accumulates BEST-TRAIN in a list of trains for output, DEC-TRAIN (initially empty).

The loop terminates when REM-TRAINS becomes empty. At this point DEC-TRAINS contains trains
with mutually exclusive sets of pulses, and therefore it can be returned as the result of DECIDE:

```
(defun decide (trains)
       (do ((rem-trains trains
                          (remove-pulses best-train
                                                 rem-trains))
            (dec-trains nil (cons best-train dec-trains))
            (best-train))
           ((not rem-trains)
            dec-trains)
           (setq best-train (append (best rem-trains) nil))))
```

---

*Contrary to first appearances, REMOVE-PULSES contains no knowledge of the structure of a train. All such knowledge needed
by REMOVE-PULSES is contained within the call to TRAIN-MINUS.
†Since REMOVE-PULSES (which uses TRAIN-MINUS, which in turn uses DELQ) will be used to resolve the conflicts, the
train is actually copied with APPEND before assignment to prevent accidents.

13

The function BEST finds the "best" train in its argument TRAINS by simply making a pass through TRAINS keeping track of the best train seen so far:

```
(defun best (trains)
       (do ((this-train (car trains) (car rem-trains))
            (rem-trains (cdr trains) (cdr rem-trains))
            (best-tr nil (better-of best-tr this-train)))
           ((not this-train) best-tr)))
```

The comparison between trains is done with the function BETTER-OF:

```
(defun better-of (train1 train2)
       (let ((n1 (nr-pulses train1))
             (n2 (nr-pulses train2)))
            (cond ((> n1 n2) train1)
                  ((> n2 n1) train2)
                  (t train1))))
```

BETTER-OF bases its judgment strictly on the numbers of pulses in the two trains, returning the longer. A better version might use the same kinds of train parameters discussed earlier with respect to QUAL-TRAIN.

In some sense BETTER-OF and QUAL-TRAIN are redundant, and one function could probably be designed to serve both purposes. With the very simple length-based discriminant used here, however, they have slightly different requirements. BETTER-OF is tasked with always indicating the best train, even when the differences in "quality" are slight. The two arguments are exactly interchangeable. QUAL-TRAIN, although a function of one argument, certainly could have been written as a function of two arguments by including the comparison now performed in OUTPUT-AGED. Those two arguments are not interchangeable, however. One is an old train on the verge of being output. The other is a current train which will be kept in the system for further updating. QUAL-TRAIN must indicate that the current train is superior to the old train *only* when there is a clear preference. In other words, since it is generally desirable that an old train be output intact whenever possible, it is given the benefit of the doubt in the comparison.

This completes the description of the Sorter. Appendix A gives the definitions of some functions used for testing, and Appendix B gives some examples showing the details of how some typical sorts develop.

PERFORMANCE OF THE PASCAL VERSION

The experimental data used to test the PASCAL version of the Sorter, referred to just as the "Sorter" for the remainder of this section, originated in an experiment conducted at NRL's Chesapeake Bay Detachment (CBD). A receiver monitoring a band of frequencies in the 9-GHz region was set up to record certain data on magnetic tape for each pulse received, including time of arrival to the nearest microsecond. A test aircraft was flown carrying a radar transmitter with known characteristics. The data collected consisted of numerous pulse trains both from the test aircraft and from numerous "targets of opportunity," consisting mostly of small surface search radars belonging to vessels on the Chesapeake Bay.

When the trains output by the Sorter were printed for examination, it was found that most trains were sorted perfectly. (The Sorter, as shown in Appendix C, outputs only statistical summaries of each train. Versions used in this early testing printed out each sorted train in detail, showing each pulse.)

No instances of a pulse from one train being put into another by the Sorter were noted. The errors made by the Sorter involved showing what was actually one train as two or more trains. For example, if the pulses in a train are considered as numbered chronologically, the Sorter might typically put the odd-numbered pulses into one train and the even-numbered pulses into another. Upon close examination of the times associated with the pulses, the trains that were being split in this fashion usually had pulse repetition intervals that alternated between two slightly different values. Other instances of splitting occurred in which there was less structure in the resulting split. In these cases the actual trains usually had interpulse intervals that were not very stable and showed no particular pattern. Perhaps some adjustment to PERFECT-FIT or FITS-OK could improve the sorting in these circumstances. For our purposes it was not worth pursuing.

Statistical information on the operation of the Sorter was gathered on one particular run. The results are shown in Table 2. The quantity "average number of trains in the system" refers to the number of trains in the variable TRAINS at the time a new pulse is obtained from GET-PULSE. This quantity, together with the item "fraction of pulses not a perfect fit," indicates that the burden imposed by the splitting of trains is not excessive.

Table 2 — Typical Sorter Performance

Duration of data collection	10 min
Number of pulses sorted	3264
Number of different transmitters present	2
Fraction of pulses not a perfect fit	18 %
Average number of trains in the system	4.9
Average number of pulses per sorted train	10.7
Total pulse data storage used	1300 bytes
Total list storage used	10540 bytes
Approximate ASC CP time to sort	35.8 s

SUMMARY

A program which does PRF sorting using list processing techniques has been described. The detailed discussion of the operation of the program was based on a LISP implementation that functions on a demonstration level. A PASCAL version capable of operation on real data is given in an appendix. The operation of the PASCAL version parallels the operation of the LISP version except as described in comments in the PASCAL listing.

A unique feature of the operation of this particular sorting algorithm is the ability to postpone a decision about whether a pulse should be assigned to a particular pulse train by duplicating or splitting the train and assigning the pulse to one of the two resulting pulse trains. The particular type of list structure used to represent pulse trains, native to LISP, prevents the splitting from resulting in unreasonable storage requirements. The necessary list-manipulation facilities have been duplicated in the PASCAL version.

15

REFERENCES

1. G.L. Steele, Jr., and G.J. Sussman, "Design of a LISP-Based Microprocessor," Commun. ACM **23** (11), 628-645 (Nov. 1980).

2. G.J. Sussman, J. Holloway, G.L. Steele, Jr., and A. Bell, "Scheme-79—Lisp on a Chip," Computer **14** (7), 10-21 (July 1981).

3. P.H. Winston, and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Mass., 1981.

4. L. Siklossy, *Let's Talk LISP*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

5. P.H. Winston, *Artificial Intelligence*, Addison-Wesley, Reading, Mass., 1977.

6. Jon L. White, File MC:LISP;LISP NEWS on the MACSYMA Consortium PDP-10 at Massachusetts Institute of Technology, Cambridge, Mass., Jan. 27, 1979.

7. D.A. Moon, *MACLISP Reference Manual*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., Mar. 6, 1976.

Appendix A
SOME FUNCTIONS TO SUPPORT SORTER TESTING

```lisp
;;;; First change some details of the LISP environment:

(*nopoint t)                        ; no decimal point on integer output

(sstatus feature noldmsg)       ; no message on autoloading

(setq prin1 'sprin1)    ; causes LISP output to be  "prettyprinted"

(setq  base 10.)        ; set default radix for numerical I/O to 10

(setq ibase 10.)


;;;; The testing functions:

(defun train (start pri n)
       (cond ((equal 0. n) nil)
             (t (cons start
                      (train (+ start pri) pri (sub1 n))))))


(defun trains (specs)
       (cond ((not (cdr specs))
              (apply 'train (car specs)))
             (t (merge2 (apply 'train (car specs))
                        (trains (cdr specs))))))


(defun merge2 (list1 list2)
       (cond ((not list2) list1)
             ((not list1) list2)
             ((lessp (car list1) (car list2))
              (cons (car list1)
                    (merge2 (cdr list1) list2)))
             (t (cons (car list2)
                      (merge2 list1 (cdr list2))))))


(defun details (pulses)
       (trace ((sort put-pulse) arg)
              ((get-pulse update groups) value))
       (let ((out (sort pulses)))
            (untrace)
            out))
```

17

Appendix B
EXAMPLES OF THE SORT PROCESS

Two examples of the sort process are shown below. The function DETAILS (Appendix A) is used to set up a trace prior to invoking SORT. The trace shows the arguments to SORT and PUT-PULSE and the values returned by GET-PULSE, UPDATE, and GROUPS. The final result of the sort follows the end of the trace. First, a simple example:

```
(details (train 1 5 5))

(1 ENTER SORT ((1 6 11 16 21)))
    (1 EXIT GET-PULSE 1)
    (1 EXIT UPDATE ((1)))
    (1 EXIT GET-PULSE 6)
    (1 EXIT UPDATE ((6) (6 1) (1)))
    (1 EXIT GET-PULSE 11)
    (1 EXIT UPDATE ((1) (11 6 1) (6)))
    (1 EXIT GET-PULSE 16)
    (1 EXIT UPDATE ((16 11 6 1) (6)))
    (1 EXIT GET-PULSE 21)
    (1 EXIT UPDATE ((21 16 11 6 1)))
    (1 EXIT GET-PULSE NIL)
  (1 EXIT GROUPS ((21 16 11 6 1)))
    (1 EXIT GET-PULSE NIL)
  (1 EXIT GROUPS NIL)
((21 16 11 6 1))
```

The example just given is not very interesting because there is no possible doubt about what the desired outcome is. The following example calls SORT with a sequence of pulses that can be separated into trains in several reasonable ways:

```
(details (trains '((1 5 4) (2 5 5) (13 3 6))))

(1 ENTER SORT ((1 2 6 7 11 12 13 16 16 17 19 22 22 25 28)))
    (1 EXIT GET-PULSE 1)
    (1 EXIT UPDATE ((1)))
    (1 EXIT GET-PULSE 2)
    (1 EXIT UPDATE ((2) (2 1) (1)))
    (1 EXIT GET-PULSE 6)
    (1 EXIT UPDATE ((6) (6 1) (6 2) (2) (2 1) (1)))
    (1 EXIT GET-PULSE 7)
    (1 EXIT UPDATE ((7) (7 6) (7 2) (7 1) (1) (2) (6 2) (6 1) (6)))
    (1 EXIT GET-PULSE 11)
    (1 EXIT UPDATE ((6) (11 6 1) (6 2) (2) (1) (7 1) (7 2) (7 6) (7)))
    (1 EXIT GET-PULSE 12)
    (1 EXIT UPDATE ((7) (12 7 2) (7 1) (11 6 1) (6)))
    (1 EXIT GET-PULSE 13)
    (1 EXIT UPDATE ((6) (11 6 1) (13 7 1) (12 7 2) (7)))
    (1 EXIT GET-PULSE 16)
    (1 EXIT UPDATE ((7) (12 7 2) (13 7 1) (16 11 6 1) (6)))
    (1 EXIT GET-PULSE 16)
```

```
    (1 EXIT UPDATE ((16) (12 7 2) (13 7 1) (16 11 6 1)))
    (1 EXIT GET-PULSE 17)
    (1 EXIT UPDATE ((16 11 6 1) (13 7 1) (17 12 7 2) (16)))
    (1 EXIT GET-PULSE 19)
    (1 EXIT UPDATE ((16) (17 12 7 2) (19 13 7 1) (16 11 6 1)))
    (1 EXIT GET-PULSE 22)
    (1 EXIT UPDATE ((16 11 6 1) (19 13 7 1) (22 17 12 7 2) (16)))
    (1 EXIT GET-PULSE 22)
    (1 ENTER PUT-PULSE (22))
  (1 EXIT GROUPS ((16 11 6)))
    (1 EXIT GET-PULSE 22)
    (1 EXIT UPDATE ((22) (22 17 12 7 2) (19 13 7 1)))
    (1 EXIT GET-PULSE 25)
    (1 EXIT UPDATE ((25 19 13 7 1) (22 17 12 7 2) (22)))
    (1 EXIT GET-PULSE 28)
    (1 EXIT UPDATE ((28) (28 22) (25 19 13 7 1) (22 17 12 7 2) (22)))
    (1 EXIT GET-PULSE NIL)
    (1 ENTER PUT-PULSE (NIL))
  (1 EXIT GROUPS ((22 17 12 2)))
    (1 EXIT GET-PULSE NIL)
  (1 EXIT GROUPS ((28) (25 19 13 7 1)))
    (1 EXIT GET-PULSE NIL)
  (1 EXIT GROUPS NIL)
((25 19 13 7 1) (28) (22 17 12 2) (16 11 6))
```

Appendix C
PASCAL VERSION OF THE SORTER

PASCAL PROGRAM

```
program lisp( input, output, summarys );
```

{ PRFSORT
  vers 1.7,
  operational 1/15/81,
  last mod 3/13/81, Friday
  language: PDL2, an extended dialect of PASCAL,
  J. O. Coleman, NRL Code 5312   }

{ Vers 1.7.1 differs from 1.7 only in that the comment describing the
  modifications associated with different versions has been deleted
  and this comment substituted in its place.   }

{ INPUT control file:

    The first line of the input file should contain an integer
representing the maximum number of pulses to be sorted.   This
quantity defaults to 5000 if no input file is present.   The
remainder of the input file, if present, restricts the sort by
specifying bounds in time, bearing, and amplitude. A pulse must
fall within all three bounds simultaneously to be forwarded to the
Sorter. A set of bounds consists of two input lines, the first
giving lower bounds, and the second giving upper bounds.   Each
line contains three numbers: time in microseconds (integer 0 to
2047999999), bearing in degrees (real 0.0 to 360.0), and
amplitude in volts (real 0.0 to 5.0). The inputs are not checked
for validity. Time is measured from the most recent multiple of
2,048,000,000 microseconds past the hour. This quantity is 34
minutes, 8 seconds. This effectively divides each clock hour into
two windows, each of which has its own sets of bounds. A blank
line is used to precede the first set and to separate successive
sets of bounds in the input file.   A line with the letter c in
column 1 in place of a set of bounds signals the switch to the
next time window. Sets of bounds should not overlap in time.   If
data exist beyond the last set of bounds, all of that data will be
accepted.

    As an example, suppose the file DATA contains data from times
ranging from 12:30:00 to 1:05:00. The input file might look like

/ start acnm=input
  1000000

  0   50.0   0.0
  2047999999 130.0   5.0

c

  0   50.0   4.0
      60000000 130.0   5.0

```
     120000000  50.0    4.0
     180000000 130.0    5.0

c
/ stop
```

In this example a maximum of one million pulses will be accepted.
Bounds are put on all data from times prior to 1:00:00. All
pulses from after 1:00:00 are accepted by default, since bounds
are given for only two of three windows. (Note that an empty file
would accept all data by default.) Prior to 1:00:00 data is
accepted only if it has bearing between 50 and 130 degrees. In
the second window data is accepted only in the first and third
minutes, and then only if it also has amplitude of between 4 and 5
volts.

Note that real quantities must have a decimal and at least one
digit on each side of the decimal. Integers must have no decimal.
Placement on the line is unimportant, except that numbers must be
separated by at least one space. The blank lines need not in fact
be blank, as they will be ignored. They can be used for comments.
The area to the right of the space to the right of the last used
number on each line is similarly ignored, as is the remainder of
the line following the c in column 1.
}

```
const max_train_length    = 50 ;       { used in OLD          }
      max_output_line      = 80 ;       { used in PRINTF       }
      default_max_pulses = 5000 ;       { used in SORT         }

   { units of microseconds: }
      max_pri              = 10000 ;    { used in OLD          }
      jitter_small         = 5 ;        { used in PERFECT_FIT }
      jitter_large         = 15 ;       { used in OLD          }

      pi = 3.14159 ;
      maxmicroseconds = maxint ;
      toa_cycle = 2048000000 ;
      hour_cycle = (3600 - 2048) * 1000000 ;

type radians = real ;
     volts   = real ;
     microseconds = integer ;

     summary = record
                   n : integer ;
                   time : record
                              earliest,
                              latest : microseconds
                          end ;
                   pri,
                   bearing,
                   amplitude : record
                                   case { accumulating } boolean of
                                       true :
                                       ( sum,
```

```
                                        sumsq : real ) ;
                                 false :
                                  ( mu,
                                    sigma : real )
                        end
             end ;

  { llllllllll  lisp kernel  llllllllll }

    s_exp = ^ s_exp_rec ;

    s_exp_rec = record
                    ref_count : integer ;
                    atomic : boolean ;
                    case boolean of
                      { selector corresponds to field atomic
                        except for nil_ ^   }
                      true  :
                       (

  { uuuuuuuuuu  user specified atomic field list:   uuuuuuuuuu }

   toa_  : microseconds ;
   bearing : radians ;
   amplitude : volts

  { llllllllll  more lisp kernel:  llllllllll }

                         ) ;
                         false :
                          ( cdr_  : s_exp ;
                            case  { deallocating }  boolean of
                                true : ( previous : s_exp ;
                                         the_car : boolean ) ;
                                false :( car_  : s_exp ) )
                    end ;

var garbage : record
                    lists,  { ref_count not maintained
                             in garbage.lists  }
                    atomic : s_exp
               end ;
    conses_created,
    atoms_created : integer ;
    t, nil_, setq_temp : s_exp ;

  { uuuuuuuuuu  user global variables:  uuuuuuuuuu }

   last_cycle : ( at_34_08, at_hour ) ;   { see cycle, summarize }
   dummy, from_file : s_exp ;

  { llllllllll more lisp kernel:  llllllllll }

          {  The following discussion applies ONLY to variables
             of type S_EXP.
```

22

Reference count maintainance:

Except for the circumstances described immediately below, all assignments to variables should be made with SETQ, not := .

With the exception of kernel routines optimized for speed, every function or procedure parameter should have REF called on it at the beginning of the body. Every local variable should be intialized to NIL_ (:= ,not SETQ) at the beginning of the body. For functions a local variable should be used to hold the function value until the end of the body. At the end of the body, the function value should be assigned to the function name (:= , not SETQ). Following that, DEREF should be called on each of the parameters and each of the local variables, with the exception of the local that held the function value at the time it was assigned to the function name. TEMP_REF should be called on that local as the last executable code in the body. }

```
macro ref( a ) = " if a <> nil_ then
                     with a ^_do
                        ref_count := ref_count + 1 " ;

      deref( a ) = " if a <> nil_ then
                       with a ^_do
                          begin
                          ref_count := ref_count - 1 ;
                          if ref_count <= 0 then
                             { non-std PASCAL use of with: }
                             deallocate( a )
                          end " ;

      temp_ref( a ) = " if a <> nil_ then
                          with a ^_do
                             ref_count := ref_count - 1 " ;

      setq( a, b ) = " begin
                       setq_temp := b ;
                       ref( setq_temp ) ;
                       deref( a ) ;
                       a := setq_temp
                       end " ;

   {  The CAR/CDR family of macros should be called  only
      with variable (not  general expression)  arguments.
      If they need to be used with more general arguments
      (say, a function call), they should be rewritten as
      functions.  }

   car( a ) = " a ^. car_ " ;

   cdr( a ) = " a ^. cdr_ " ;
```

```
    caar( a ) = " car( car( a ) ) " ;

    cdar( a ) = " cdr( car( a ) ) " ;

    cadr( a ) = " car( cdr( a ) ) " ;

    cadar( a ) = " car( cdr( car( a ) ) ) " ;

{ uuuuuuuuuu user macros uuuuuuuuuu }

  { TOA should be called only on arguments which are
    variables or combinations of car and cdr of variables.  }

 toa( pulse ) = " pulse ^. toa_ " ;

 init_train( pulse ) = " cons( pulse, nil_ ) " ;

 add_pulse( pulse, train ) = " cons( pulse, train ) " ;

 nr_pulses( train ) = " length( train ) " ;

 latest_pulse( train ) = " car( train ) " ;

 next_latest_pulse( train ) = " cadr( train ) " ;

{ llllllllll  more lisp kernel: llllllllll }

function cons( a, d : s_exp ) : s_exp ;
   var out : s_exp ;

   begin  { cons }
   { get a cons in OUT: }
   if garbage.lists = nil_  then
         begin
         new( out, false ) ;
         out ^. atomic := false ;
         conses_created := conses_created + 1
         end
      else
         begin    { see macro FREE_CONS in DEALLOCATE }
         out := garbage.lists ;
         garbage.lists := garbage.lists ^. previous
         end ;

   ref(a) ;
   ref(d) ;
   with out ^ do
      begin
      car_ := a ;
      cdr_ := d ;
      ref_count := 0     { fn values returned are temp_ref }
      end ;
   cons := out
   end ;  { cons }

function stack_length( l : s_exp ) : integer ;
```

24

```
    var n : integer ;

    begin  { stack_length }
    n := 0 ;
    while l <> nil_ do
        begin
        n :=n + 1 ;
        l := l ^. previous
        end ;
    stack_length := n
    end ;  { stack_length }

procedure deallocate( current : s_exp ) ;

    label 1, 2, 3, 4 ;

    const done = nil ;  { UNIQUE empty-stack marker }

    var from,           { linked stack of values of CURRENT with each
                          stacked cons containing the associated
                          "return address" flag in the field THE_CAR }

        next : s_exp ;  { temporary }

    macro start      = "1" ;        { mnemonics for labels }
          done_car   = "2" ;
          done_cdr   = "3" ;
          pop_stack  = "4" ;

          traverse_sub( field, field_is_car_ ) = "
              begin
              assert not current ^. atomic ;
              next := current ^. field ;     { this next is local }
              if next <> nil_ then           {    to this macro   }
                  with next ^ do
                      begin
                      ref_count := ref_count - 1 ;
                      if ref_count <= 0 then
                          begin  { switch to TRUE variant
                                   of non-atomic s_exp_rec }
                          with current ^ do
                              begin
                              the_car := field_is_car_ ;
                              previous := from
                              end ;
                                      { note PREVIOUS is defined as the
                                        first of car_ or cdr_ traversed }
                          from := current ;
                          current := next ;
                          goto start
                          end
                      end
              end " ;

          free_atom( a ) = "
              begin
```

```
            next := cons( a, garbage.atomic ) ;
            assert next ^. ref_count = 0 ;
            next ^. ref_count := 1 ;
            if garbage.atomic <> nil_ then
                with garbage.atomic ^ do
                    ref_count := ref_count - 1 ;
            garbage.atomic := next
            end " ;

        free_cons( c ) = "
            begin
            c ^. previous := garbage.lists ;
                    { note PREVIOUS must represent
                      same field in function CONS }
            garbage.lists := c
            end " ;

  { This procedure is equivalent to the following recursive version,
    differing only in that the recursion is removed by using a stack
    embedded in the structure being deallocated:

   begin
   if current ^. atomic then
        free_atom( current )
      else
        begin
        deref (current ^. car_) ;
        deref (current ^. cdr_) ;
        free_cons( current )
        end
   end ;
  }

   begin  { deallocate }
   from := done ;                          { initialize stack }

start:
  assert current <> nil_ ;
  assert current ^. ref_count <= 0 ;

  if current ^. atomic then
     begin
     free_atom( current ) ;
     goto pop_stack
     end ;

  traverse_sub( car_, true ) ;

done_car:
    { assert finished with subtree current ^. car_ }

  traverse_sub( cdr_, false ) ;

done_cdr:
    { assert finished with both subtrees }
  assert current ^. ref_count <= 0 ;
```

26

```
        assert not current ^. atomic ;

        free_cons( current ) ;

   pop_stack:
        { assert current is deallocated }

        if from <> done then
           begin
           current := from ;
           from := from ^. previous ;
           if current ^. the_car then
                goto done_car
              else
                goto done_cdr
           end
        end ;   { deallocate }

function length( l : s_exp ) :integer ;
   var len : integer ;
       this : s_exp ;

   begin  { length }
   ref( l ) ;
   len := 0 ;
   this := l ;
   while this <> nil_ do
      begin
      len := len + 1 ;
      this := cdr( this )
      end ;
   length := len ;
   deref( l )
   end ;   { length }

function get_atom : s_exp ;
   var out : s_exp ;

   begin  { get_atom }
   out := nil_ ;
   if garbage.atomic = nil_ then
         begin
         new( out, true ) ;
         out ^. atomic := true ;
         out ^. ref_count := 1 ;
         atoms_created := atoms_created + 1
         end
      else
         begin    { see macro FREE_ATOM in DEALLOCATE }
         setq( out, car( garbage.atomic ) ) ;
         setq( garbage.atomic, cdr( garbage.atomic ))
         end ;
   get_atom := out ;
   temp_ref( out )
   end ;   { get_atom }
```

27

```
function append( x, y : s_exp ) : s_exp ;

  { Derived from the original recursive version shown  following
    this function.  A proof of correctness has been carried out.  }

    var last, stack, temp : s_exp ;

    macro empty = "nil_" ;    { unique flag }
                              { must be valid 2nd arg to CONS }
    begin  { append }
    if x = nil_ then
          append := y
      else
          begin
          assert not x ^. atomic ;
          ref( x ) ;

          stack := empty ;
          temp := x ;
          repeat
             stack := cons( car( temp ), stack ) ;
             temp := cdr( temp )
          until temp = nil_ ;

          ref( y ) ;
          with stack ^ do
             begin    { new use of TEMP }
             temp := cdr_ ;
             cdr_  := y ;
             if temp <> empty then
                 begin
                 ref_count := 1 ;
                 repeat
                    last := stack ;
                    stack := temp ;
                    temp := cdr( stack ) ;
                    stack ^. cdr_ := last
                 until temp = empty ;
                 stack ^. ref_count := 0
                 end ;
             append := stack
             end ;
          deref( x )
          end
    end ;  { append }

  { the recursive version used previously is
function append( x, y : s_exp ) : s_exp ;
    var temp : s_exp ;
    begin
    ref( x ) ;
    ref( y ) ;
    temp := nil_ ;
    if x = nil_ then
          setq( temp, y )
      else
```

```
              setq( temp, cons( car( x ), append( cdr( x ), y ) ) ) ;
      append := temp ;
      deref( x ) ;
      deref( y ) ;
      temp_ref( temp )
      end ;
}

   function delq( x, y : s_exp ) : s_exp ;
      var out, this, last : s_exp ;
      begin   { delq }
      ref( x ) ;
      ref( y ) ;
      out := y ;
      if y <> nil_ then
          begin
          assert not y ^. atomic ;
          with y ^ do
              if x = car_ then
                  out := cdr_
              else
                  begin
                  this := cdr_ ;
                  last := y ;
                  while this <> nil_ do
                      with this ^ do
                          if x = car_ then
                                  begin
                                  this := cdr_ ;
                                  ref( this ) ;   { 3 lines like setq }
                                  deref( cdr( last ) ) ;
                                  last ^. cdr_ := this
                                  end
                              else
                                  begin
                                  last := this ;
                                  this := cdr_
                                  end
                  end
          end ;
      ref( out ) ;
      deref( x ) ;
      deref( y ) ;
      temp_ref( out ) ;
      delq := out
      end ;   { delq }

   function memq( x, y : s_exp ) : s_exp ;
      label 1;
      var temp, out : s_exp ;
      macro exit = "1" ;   { mnemonic }

      begin   { memq }
      ref( x ) ;
      ref( y ) ;
      out := nil_ ;
```

29

```
      temp := y ;
      while temp <> nil_ do
         with temp ^ do
            if car_ = x then
                  begin
                  ref( temp );
                  out := temp ;
                  goto exit
                  end
              else
                  temp := cdr_ ;
  exit:
    memq := out ;
    deref( x ) ;
    deref( y ) ;
    temp_ref( out )
    end ;  { memq }


function reverse( 1 : s_exp ) : s_exp ;
   var out : s_exp ;

   begin  { reverse }
   ref( 1 ) ;
   out := nil_ ;
   if 1 <> nil_ then
      setq( out, append( reverse( cdr( 1 ) ),
                         cons( car( 1 ), nil_ ) ) ) ;
   reverse := out ;
   deref( 1 ) ;
   temp_ref( out )
   end ;  { reverse }


function printf( obj : s_exp ) : s_exp ;

    { PRINTF passes its argument along as its
      value, printing it along the way. }

   var line_length :  0..max_output_line  ;

   procedure newline ;
      begin  { newline }
      writeln ;
      write( ' ' ) ;
      line_length := 0
      end ;  { newline }

   procedure stringout( string : packed array[0..?]of char ) ;
      var string_length : 0..max_output_line ;
      begin
      string_length := ub(string,1) - lb(string,1) + 1 ;
      if line_length + string_length > max_output_line then
         newline ;
      write( string ) ;
      line_length := line_length + string_length
      end ;  { stringout }
```

```
procedure real_nr_out( real_nr : real ; width, dec : integer ) ;
    begin
    if line_length + width > max_output_line then
        newline ;
    write( real_nr : width : dec ) ;
    line_length := line_length + width
    end ;   { real_nr_out }

procedure usec_out( usec : microseconds ; width : integer ) ;
    begin
    if line_length + width > max_output_line then
        newline ;
    write( usec : width ) ;
    line_length := line_length + width
    end ;   { usec_out }

procedure obj_print( obj : s_exp ) ;

    procedure atom_print( a : s_exp ) ;
        begin  { atom_print }
        newline ;
        if a = nil_ then
            stringout(' nil ')
        else if a = t then
            stringout(' t ')
        else
            with a^ do
                begin
{ uuuuuuuuuu user's atom printout  uuuuuuuuuu }

                stringout( ' [' ) ;
                usec_out( toa_, 12 ) ;
                stringout( ' usec.,' ) ;
                real_nr_out( (180.0/pi) * bearing, 7, 2 ) ;
                stringout( ' deg.,' ) ;
                real_nr_out( amplitude, 6, 3 ) ;
                stringout( ' volts ]' )

{ llllllllll  more lisp kernel:  llllllllll }
                end
        end ;   { atom_print }

    begin  { obj_print }
    if obj^.atomic then
        atom_print( obj )
        else
            begin
            newline ;
            stringout( ' (' ) ;
            while obj <> nil_ do
                begin
                obj_print( obj^.car_ ) ;
                obj_ := obj^. cdr_
                end ;
            stringout( ' )' )
            end
```

31

```
      end ;   { obj_print }

   begin  { printf }
   newline ;
   obj_print( obj ) ;
   newline ;
   printf := obj
   end ;   { printf }

 { uuuuuuuuuu  user functions, procedures: uuuuuuuuuu }

function iabs( i : integer ) : integer ;

   begin  { iabs }
   if i < 0 then
         iabs := - i
      else
         iabs := i
   end ;   { iabs }

function next_filed_pulse : s_exp ;

   { Builds an atom to represent a pulse, interpreting
     integer data fields from the file DATA as relevant quantities }

   const    ok = 0 ;          { see asc subprogram library.. }
            eof = #8100 ;    { ..notebook page i9.000 re: word6 }
            df = 1 ;
            two_9th = #200 ;

   var out : s_exp ;
       eastv, northv : volts ;
       status, source : integer ;
       data : array [ 1..4 ] of integer ;

   procedure unpack( var status, source : integer ;
                     var data : array [ 1..4 ] of integer ) ;
      fortran ;  { see RATFOR section }

   function atan2( var y, x : volts ) : radians ; fortran ;

   begin  { next_filed_pulse }
   out := nil_ ;

   repeat
      unpack( status, source, data ) ;
   until( source = df ) or ( status = eof ) ;

   if status <> eof then
      begin
      assert ( status = ok ) and ( source = df ) ;
      setq( out, get_atom ) ;
      with out ^ do
         begin
         eastv := ( ( two_9th - data[ 3 ] ) * 5.0 ) / two_9th ;
         northv := - ( ( two_9th - data[ 4 ] ) * 5.0 ) / two_9th ;
```

32

```
            amplitude := sqrt( eastv * eastv + northv * northv ) ;

            if ( eastv = 0.0 ) and ( northv = 0.0 ) then
                 bearing := 0.0
               else
                 bearing := atan2( eastv, northv ) ;

            if bearing < 0.0 then
                 bearing := bearing + 2.0 * pi ;

            toa_ := ( data[ 1 ] mod 2048 ) * 1000000 + data[ 2 ] ;
            end
         end ;

     next_filed_pulse := out ;
     temp_ref( out )
     end ;  { next_filed_pulse }

function better_of( train1, train2 : s_exp ) : s_exp ;
   var n2, n1 : integer ;
       out : s_exp ;

   begin  { better_of }
   ref( train1 ) ;
   ref( train2 ) ;
   out := nil_ ;

   n1 := nr_pulses( train1 ) ;
   n2 := nr_pulses( train2 ) ;

   if n1 >= n2 then
      setq( out, train1 )
   else
      setq( out, train2 ) ;

   better_of := out ;
   deref( train1 ) ;
   deref( train2 ) ;
   temp_ref( out )
   end ;  { better_of }

function qual_train( train : s_exp ) : integer ;

   begin  { qual_train }
   ref( train ) ;

   assert not train ^. atomic ;

   with train ^ do
      if cdr_ = nil_ then
         qual_train := 1
      else if cdr( cdr_ ) = nil_ then
         qual_train := 2
      else
         qual_train := 3 ;
```

33

```
        deref( train )
        end ;  { qual_train }

  function perfect_fit( pulse, train : s_exp ) : s_exp ;

   { Detects the perfect fit of  a pulse to a  train.  A perfect fit  is
     declared if the  time of  the pulse is  within either  JITTER_SMALL
     microseconds of the  predicted time  (predicted from  the last  two
     pulses in the train) or within 0.5 % of the prediction interval  of
     the predicted time. }

        var out_temp : s_exp ;
        error, last_time, last_pri : microseconds ;

        begin  { perfect_fit }
        ref( pulse ) ;
        ref( train ) ;
        out_temp := nil_ ;

        if cdr( train ) = nil_  then
            setq( out_temp, nil_ )
          else
            begin
            last_time := toa( latest_pulse( train ) ) ;
            last_pri := last_time - toa( next_latest_pulse( train ) ) ;

            error := iabs( toa( pulse ) - last_time - last_pri ) ;

            if ( error <= jitter_small )
               or ( error / last_pri <= 0.005 ) then
                  setq( out_temp, add_pulse( pulse, train ) )
               else
                  setq( out_temp, nil_ )
            end ;

        perfect_fit := out_temp ;
        deref( pulse ) ;
        deref( train ) ;
        temp_ref( out_temp )
        end ;  { perfect_fit }

  function train_minus( a, b : s_exp ) : s_exp ;
        var temp38,  up_a, rem_b, this_b  : s_exp ;

        begin  { train_minus }
        ref( a ) ;
        ref( b ) ;
        temp38 := nil_ ;
        up_a := nil_ ;
        rem_b := nil_ ;
        this_b := nil_ ;

        if a = nil_  then
            setq( temp38, nil_ )
          else
            begin
```

```
            setq( this_b, car( b ) ) ;
            setq( rem_b, cdr( b ) ) ;
            setq( up_a, a ) ;
            while this_b <> nil_ do
                begin
                setq( up_a, delq( this_b, up_a ) ) ;
                setq( this_b, car( rem_b ) ) ;
                setq( rem_b, cdr( rem_b ) )
                end ;
            setq( temp38, up_a )
            end ;

    train_minus := temp38 ;
    deref( a ) ;
    deref( b ) ;
    deref( up_a ) ;
    deref( rem_b ) ;
    deref( this_b ) ;
    temp_ref( temp38 )
    end ;   { train_minus }

function remove_pulses( best_train, trains : s_exp ) : s_exp ;
    var stripped, temp42, up_trains, rem_trains,
        this_train : s_exp ;

    begin  { remove_pulses }
    ref( best_train ) ;
    ref( trains ) ;
    temp42 := nil_ ;
    stripped := nil_ ;
    up_trains := nil_ ;
    rem_trains := nil_ ;
    this_train := nil_ ;

    setq( this_train, car( trains ) ) ;
    setq( rem_trains, cdr( trains ) ) ;
    setq( up_trains, nil_ ) ;
    while this_train <> nil_ do
        begin
        setq( stripped, train_minus( this_train, best_train ) ) ;
        if stripped <> nil_ then
            if memq( stripped, up_trains ) = nil_ then
                setq( up_trains, cons( stripped, up_trains ) ) ;
        setq( this_train, car( rem_trains ) ) ;
        setq( rem_trains, cdr( rem_trains ) )
        end ;
    setq( temp42, up_trains ) ;

    remove_pulses := temp42 ;
    deref( best_train ) ;
    deref( trains ) ;
    deref( stripped ) ;
    deref( up_trains ) ;
    deref( rem_trains ) ;
    deref( this_train ) ;
    temp_ref( temp42 )
```

```
    end ;   { remove_pulses }

function best( trains : s_exp ) : s_exp ;
    var temp46, temp47, best_tr, rem_trains, this_train : s_exp ;

    begin  { best }
    ref( trains ) ;
    temp46 := nil_ ;
    temp47 := nil_ ;
    best_tr := nil_ ;
    rem_trains := nil_ ;
    this_train := nil_ ;

    setq( this_train, car( trains ) ) ;
    setq( rem_trains, cdr( trains ) ) ;

    if this_train <> nil_ then
         setq( temp47, nil_ )
      else
         setq( temp47, t ) ;

    while temp47 = nil_ do
      begin
      setq( best_tr, better_of( best_tr, this_train ) ) ;
      setq( this_train, car( rem_trains ) ) ;
      setq( rem_trains, cdr( rem_trains ) ) ;

      if this_train <> nil_ then
           setq( temp47, nil_ )
        else
           setq( temp47, t )
      end ;
    setq( temp46, best_tr ) ;

    best := temp46 ;
    deref( trains ) ;
    deref( temp47 ) ;
    deref( best_tr ) ;
    deref( rem_trains ) ;
    deref( this_train ) ;
    temp_ref( temp46 )
    end ;   { best }

function decide( trains : s_exp ) : s_exp ;
    var temp48, temp49, best_train, dec_trains, rem_trains : s_exp ;

    begin  { decide }
    ref( trains ) ;
    temp48 := nil_ ;
    temp49 := nil_ ;
    best_train := nil_ ;
    dec_trains := nil_ ;
    rem_trains := nil_ ;

    setq( rem_trains, trains ) ;
```

```
    if rem_trains <> nil_ then
        setq( temp49, nil_ )
    else
        setq( temp49, t ) ;

while temp49 = nil_ do
    begin
    setq( best_train, append( best( rem_trains ), nil_ ) ) ;
    setq( rem_trains, remove_pulses( best_train, rem_trains ) ) ;
    setq( dec_trains, cons( best_train, dec_trains ) ) ;

    if rem_trains <> nil_ then
        setq( temp49, nil_ )
    else
        setq( temp49, t )
    end ;
setq( temp48, dec_trains ) ;

decide := temp48 ;
deref( trains ) ;
deref( temp49 ) ;
deref( best_train ) ;
deref( dec_trains ) ;
deref( rem_trains ) ;
temp_ref( temp48 )
end ;  { decide }

function fits_ok( pulse, train : s_exp ) : s_exp ;

{ Declares an OK fit of a pulse  to a train if the new time  interval
  (between PULSE and the last  pulse in TRAIN) is  within 5 % of  the
  previous interval (as  determined  from  the  last  two  pulses  in
  TRAIN). }

var temp51 : s_exp ;
    last_time, last_pri : microseconds ;

begin  { fits_ok }
ref( pulse ) ;
ref( train ) ;
temp51 := nil_ ;

if cdr( train ) = nil_ then
    setq( temp51, add_pulse( pulse, train ) )
else
    begin
    last_time := toa( latest_pulse( train ) ) ;
    last_pri := last_time - toa( next_latest_pulse( train ) ) ;

    if ( iabs( toa( pulse ) - last_time - last_pri )
        / last_pri )
        <= 0.05 then

            setq( temp51, add_pulse( pulse, train ) )
        else
            setq( temp51, nil_ )
```

37

```
            end ;

        fits_ok := temp51 ;
        deref( pulse ) ;
        deref( train ) ;
        temp_ref( temp51 )
        end ;


    function update_perfect( pulse, trains : s_exp ) : s_exp ;
        var temp55, temp59, temp58, fitted, up_trains, any_perfect,
            rem_trains, this_train : s_exp ;

        begin   { update_perfect }
        ref( pulse ) ;
        ref( trains ) ;
        temp55 := nil_ ;
        temp59 := nil_ ;
        temp58 := nil_ ;
        fitted := nil_ ;
        up_trains := nil_ ;
        any_perfect := nil_ ;
        rem_trains := nil_ ;
        this_train := nil_ ;

        if trains = nil_ then
            setq( temp55, cons( init_train( pulse ), nil_ ) )
          else
            begin
            setq( this_train, car( trains ) ) ;
            setq( rem_trains, cdr( trains ) ) ;
            setq( any_perfect, nil_ ) ;
            setq( up_trains, nil_ ) ;
            while this_train <> nil_ do
                begin
                setq( fitted, perfect_fit( pulse, this_train ) ) ;

                if any_perfect <> nil_ then
                    setq( temp58, any_perfect )
                  else
                    setq( temp58, fitted ) ;

                setq( any_perfect, temp58 ) ;

                if fitted <> nil_ then
                    setq( temp59, fitted )
                  else
                    setq( temp59, this_train ) ;

                setq( up_trains, cons( temp59, up_trains ) ) ;

                setq( this_train, car( rem_trains ) ) ;
                setq( rem_trains, cdr( rem_trains ) )
                end ;

            if any_perfect <> nil_ then
                setq( temp55, up_trains )
```

```
                else
                    setq( temp55, nil_ )
            end ;

    update_perfect := temp55 ;
    deref( pulse ) ;
    deref( trains ) ;
    deref( temp59 ) ;
    deref( temp58 ) ;
    deref( fitted ) ;
    deref( up_trains ) ;
    deref( any_perfect ) ;
    deref( rem_trains ) ;
    deref( this_train ) ;
    temp_ref( temp55 )
    end ;   { update_perfect }


function update_ok( pulse, trains : s_exp ) : s_exp ;
    var temp60, temp63, fitted, up_trains, rem_trains,
        this_train : s_exp ;

    begin   { update_ok }
    ref( pulse ) ;
    ref( trains ) ;
    temp60 := nil_ ;
    temp63 := nil_ ;
    fitted := nil_ ;
    up_trains := nil_ ;
    rem_trains := nil_ ;
    this_train := nil_ ;

    if trains = nil_ then
            setq( temp60, cons( init_train( pulse ), nil_ ) )
        else
            begin
            setq( this_train, car( trains ) ) ;
            setq( rem_trains, cdr( trains ) ) ;
            setq( up_trains, trains ) ;
            while this_train <> nil_ do
                begin
                setq( fitted, fits_ok( pulse, this_train ) ) ;
                setq( this_train, car( rem_trains ) ) ;
                setq( rem_trains, cdr( rem_trains ) ) ;

                if fitted <> nil_ then
                        setq( temp63, cons( fitted, up_trains ) )
                    else
                        setq( temp63, up_trains ) ;

                setq( up_trains, temp63 )
                end ;
            setq( temp60, up_trains )
            end ;

    update_ok := temp60 ;
    deref( pulse ) ;
```

```
        deref( trains ) ;
        deref( temp63 ) ;
        deref( fitted ) ;
        deref( up_trains ) ;
        deref( rem_trains ) ;
        deref( this_train ) ;
        temp_ref( temp60 )
        end ;   { update_ok }

    function update( pulse, trains : s_exp ) : s_exp ;
        var temp64, temp65 : s_exp ;

        begin  { update }
        ref( pulse ) ;
        ref( trains ) ;
        temp64 := nil_ ;
        temp65 := nil_ ;

        setq( temp65, update_perfect( pulse, trains ) ) ;

        if temp65 <> nil_ then
             setq( temp64, temp65 )
           else
             setq( temp64, cons( init_train( pulse ),
                                 update_ok( pulse, trains ) ) ) ;

        update := temp64 ;
        deref( pulse ) ;
        deref( trains ) ;
        deref( temp65 ) ;
        temp_ref( temp64 )
        end ;   { update }

    procedure summarize( train : s_exp ;
                         var s : summary ) ;
        var rem_train, pulse : s_exp ;
            prev_toa,
            interval, sm_interval : microseconds ;
            total_intervals, nr_intervals : integer ;

        macro next_pulse( train ) = "
                begin
                setq( pulse, car( train ) ) ;
                setq( rem_train, cdr( train ) )
                end " ;

            update_interval = "
                with pulse^ do
                   begin
                   interval := prev_toa - toa_ ;
                   prev_toa := toa_
                   end " ;

        begin  { summarize }
        ref( train ) ;
        pulse := nil_ ;
```

40

```
rem_train := nil_ ;

with s do
    begin
    if train = nil_ then
            begin
            n := 0 ;
            bearing.mu := 0.0 ;
            bearing.sigma := 0.0 ;
            amplitude.mu := 0.0 ;
            amplitude.sigma := 0.0 ;
            pri.mu := 0.0 ;
            pri.sigma := 0.0 ;
            time.earliest := 0 ;
            time.latest := 0
            end
        else
            begin
            next_pulse( train ) ;
            n := 1 ;
            if rem_train = nil_ then
                    with pulse^ do
                        begin
                        s.pri.mu := 0.0 ;
                        s.pri.sigma := 0.0 ;
                        s.bearing.mu := bearing ;
                        s.amplitude.mu := amplitude ;
                        s.bearing.sigma := 0.0 ;
                        s.amplitude.sigma := 0.0 ;
                        time.earliest := toa_ ;
                        time.latest := toa_
                        end
                else    { n > 1 }
                    begin
                    time.latest := toa( pulse ) ;
                    prev_toa := time.latest ;
                    sm_interval := maxmicroseconds ;
                    with pulse^ do
                        begin
                        s.bearing.sum := bearing ;
                        s.amplitude.sum := amplitude
                        end ;

                    repeat
                        next_pulse( rem_train ) ;
                        n := n + 1 ;
                        with bearing, pulse^ do
                            sum := sum + bearing ;
                        with amplitude, pulse^ do
                            sum := sum + amplitude ;
                        update_interval ;

                        if interval < sm_interval then
                            sm_interval := interval

                    until rem_train = nil_ ;
```

41

```
time.earliest := toa( pulse ) ;
with bearing do
   mu := sum / n ;
with amplitude do
   mu := sum / n ;

next_pulse( train ) ;
prev_toa := time.latest ;
total_intervals := 0 ;

with bearing, pulse^ do
   sumsq := sqr( bearing - mu ) ;
with amplitude, pulse^ do
   sumsq := sqr( amplitude - mu ) ;

repeat
   next_pulse( rem_train ) ;
    { note the sumsq's are used for 2nd central
      moment accumulation }
   with bearing, pulse^ do
      sumsq := sumsq + sqr( bearing - mu ) ;
   with amplitude, pulse^ do
      sumsq := sumsq + sqr( amplitude - mu ) ;
   update_interval ;
   nr_intervals := round( interval / sm_interval ) ;
   total_intervals := total_intervals + nr_intervals
until rem_train = nil_ ;

with time do
   pri.mu := (latest - earliest) / total_intervals;

with bearing do
   sigma := sqrt( sumsq / ( n - 1 ) ) ;

with amplitude do
   sigma := sqrt( sumsq / ( n - 1 ) ) ;

if n = 2 then
    pri.sigma := 0.0
   else
      begin
      setq( rem_train, cdr( train ) ) ;
      prev_toa := time.latest ;
      pri.sumsq := 0.0 ;

      repeat
         next_pulse( rem_train ) ;
         update_interval ;
         nr_intervals := round( interval
                                / sm_interval ) ;
         pri.sumsq := pri.sumsq
                      + nr_intervals
                         * sqr( interval
                                / nr_intervals
                                - pri.mu )
      until rem_train = nil_ ;
```

```
                                pri.sigma := sqrt( pri.sumsq
                                            / ( total_intervals - 1 ) ) )
                        end
                end
            end ;

        with time do
            if (earliest < 0) and (latest < 0) then
                case last_cycle of
                    at_hour:
                        begin
                        earliest := earliest + hour_cycle ;
                        latest := latest + hour_cycle
                        end ;
                    at_34_08:
                        begin
                        earliest := earliest + toa_cycle ;
                        latest := latest + toa_cycle
                        end
                end   { case }
        end ;

    deref( rem_train ) ;
    deref( pulse ) ;
    deref( train )
    end ;   { summarize }

function get_mask : s_exp ;
    var lb,ub,out : s_exp ;

    begin   { get_mask }
    lb := nil_ ;
    ub := nil_ ;
    out := nil_ ;

    while not eof do
        begin
        if input^ = 'c' then
                begin
                readln ;
                setq( out, cons( t, out ) )
                end
            else
                begin
                setq( lb, get_atom ) ;

                with lb^ do
                    begin
                    readln( toa_, bearing, amplitude ) ;
                    bearing := bearing * pi / 180.0
                    end ;

                setq( ub, get_atom ) ;

                with ub^ do
                    begin
```

43

```
            readln( toa_, bearing, amplitude ) ;
            bearing := bearing * pi / 180.0
            end ;

        setq( out, cons( cons( lb, cons( ub, nil_ ) ), out ) )
        end ;

   if not eof then
       readln

   end ;

 setq( out, reverse( out ) ) ;

 get_mask := out ;
 deref( lb ) ;
 deref( ub ) ;
 temp_ref( out )
 end ;  { get_mask }

function sort( pulses : s_exp ) : s_exp ;

  { Serves as an interface to GROUP, the real Sorter.  SORT  handles
    control  input,  and  summarizes  the  sorted  groups  in  file
    SUMMARYS.  }

  var this_group, trains, old_groups,
      out, last_pulse, pulse_mask : s_exp ;

      max_used_pulses, used_pulses : integer ;

      group_summary : summary ;

      summarys : file of summary ;

  function get_pulse : s_exp ;

    { Gets the next  available input  pulse. Uses  a buffered  pulse
      stored by  PUT_PULSE if  available.  Requires pulses  to  pass
      through masks kept in  PULSE_MASK. Uses local procedure  CYCLE
      to reduce all  times in  the system by  the appropriate  amount
      when the TOA_  fields in the  input stream cycle back to  (near)
      zero. Keeps track of total number of pulses used. }

    label 1, 2, 3 ;
    var pulse, out : s_exp ;
        del_toa : microseconds ;

    macro exit = " 1 " ;    { for labels }
          test = " 2 " ;
          next_pulse = " 3 " ;

          test_passed( field ) = "
              ( ( pulse_mask^.car_^.car_^.field
                <= out^.field )
               and ( out^.field
```

44

```
                    <= pulse_mask^.car_^.cdr_^.car_^.field ) ) " ;

procedure cycle( l : s_exp ) ;

   begin  { cycle }
   ref( l ) ;

   if l <> nil_ then
      with l^ do
         if atomic then
               if toa_ > 0 then
                  { some atoms in multiple trains }
                  case last_cycle of
                     at_hour:
                        toa_ := toa_ - hour_cycle ;
                     at_34_08:
                        toa_ := toa_ - toa_cycle
                  end  { case }
                  else   { make next else go with proper if }
            else
               begin
                 { non-std (pascal) use of with }
               cycle( car( l ) ) ;
               cycle( cdr( l ) )
               end ;

   deref( l )
   end ;  { cycle }

begin  { get_pulse }
out := nil_ ;
pulse := nil_ ;

repeat
   repeat

     next_pulse:                             { referenced twice }
       if pulses = from_file then
            .setq( out, next_filed_pulse )
          else
             begin
             setq( pulse, car( pulses ) ) ;
             setq( pulses, cdr( pulses ) ) ;
             setq( out, pulse )
             end ;

        if ( out <> nil_ ) and ( last_pulse <> nil_ ) then
           begin
           del_toa := toa( out ) - toa( last_pulse ) ;
           if del_toa < -900000000  { 15 min. }  then
              { test against 0 triggers on data flaws }
              begin
              if del_toa > - hour_cycle then
                  last_cycle := at_hour
                 else
                  last_cycle := at_34_08 ;
```

```
            cycle( trains ) ;

            while car( pulse_mask ) <> t do
               setq( pulse_mask, cdr( pulse_mask ) ) ;

            setq( pulse_mask, cdr( pulse_mask ) )
            end
         end ;

      setq( last_pulse, out ) ;

      if out = nil_ then
         begin
         setq( pulses, nil_ ) ;
         goto exit
         end ;

   test:                              { referenced once }
      if pulse_mask = nil_ then
         goto exit ;

      if car( pulse_mask ) = t then
         goto next_pulse ;

      if toa( out ) < toa( caar( pulse_mask ) ) then
         goto next_pulse ;    { failed lower bound }

      if toa( out ) > toa( cadar( pulse_mask ) ) then
         begin    { failed upper bound }
         setq( pulse_mask, cdr( pulse_mask ) ) ;
         goto test
         end

   until test_passed( bearing )
   until test_passed( amplitude ) ;

 exit:                              { referenced twice }
   used_pulses := used_pulses + 1 ;

   if used_pulses >= max_used_pulses then
      setq( pulses, nil_ ) ;

   get_pulse := out ;
   deref( pulse ) ;
   temp_ref( out )
   end ;   { get_pulse }

function put_pulse( pulse : s_exp ) : s_exp ;

 { Returns PULSE to the input stream, decrementing the count of
   pulses used accordingly. }

   var out : s_exp ;

   begin   { put_pulse }
   ref( pulse ) ;
```

46

```
    out := nil_ ;

    setq( pulses, cons( pulse, pulses ) ) ;
    setq( out, pulses ) ;
    used_pulses := used_pulses - 1 ;

    put_pulse := out ;
    deref( pulse ) ;
    temp_ref( out )
    end ;   { put_pulse }

function groups : s_exp ;
    var temp66, dummy, temp67, pulse, out_trains : s_exp ;

    function output_aged( pulse : s_exp ) : s_exp ;
        var temp37, rem_trains, this_train,
            thin_old, this_old, rem, this,
            out : s_exp ;
        time : microseconds ;

        function old( train : s_exp ) : s_exp ;

        { Declares a train to be old if either it   contains
          MAX_TRAIN_LENGTH (or more) pulses, it has gone over
          MAX_PRI microseconds without  updating, or it has two  or
          more pulses and the  train's next predicted  pulse time
          (predicted from the  last two pulses)  has been missed  by
          more than JITTER_LARGE microseconds.  }

        var last_t : microseconds ;
            out : s_exp ;
            l : integer ;

        begin  { old  }
        ref( train ) ;
        out := nil_ ;

        l := nr_pulses( train ) ;
        last_t := toa( latest_pulse( train ) ) ;

        if l >= max_train_length then
            setq( out, t )
        else if time - last_t > max_pri then
            setq( out, t )
        else if l <> 1 then
                if time - last_t
                    > jitter_large
                       + ( last_t
                            - toa( next_latest_pulse( train ) ) ) then
                   setq( out, t ) ;

        old := out ;
        deref( train ) ;
        temp_ref( out )
        end ;   { old  }
```

```
begin  { output_aged }
ref( pulse ) ;
out := nil_ ;
temp37 := nil_ ;
rem_trains := nil_ ;
this_train := nil_ ;
thin_old := nil_ ;
this_old := nil_ ;
rem := nil_ ;
this := nil_ ;

time := toa( pulse ) ;

setq( this, car( trains ) ) ;
setq( rem, cdr( trains ) ) ;
while this <> nil_ do
   begin
   if old( this ) <> nil_ then
      begin
      setq( trains, delq( this, trains ) ) ;
      setq( out_trains, cons( append( this, nil_ ),
                                out_trains ) )
      end ;

   setq( this, car( rem ) ) ;
   setq( rem, cdr( rem ) )
   end ;

setq( this_old, car( out_trains ) ) ;
setq( rem, cdr( out_trains ) ) ;
setq( thin_old, nil_ ) ;
while this_old <> nil_ do
   begin
   setq( this_train, car( trains ) ) ;
   setq( rem_trains, cdr( trains ) ) ;
   while ( this_train <> nil_ )
         and (this_old <> nil_ ) do
      begin
      if qual_train( this_old )
         <= qual_train( this_train ) then

         setq( this_old,
               train_minus( this_old, this_train ) ) ;

      setq( this_train, car( rem_trains ) ) ;
      setq( rem_trains, cdr( rem_trains ) )
      end ;

   setq( trains, remove_pulses( this_old, trains ) ) ;

   if this_old <> nil_ then
        setq( temp37, cons( this_old, thin_old ) )
      else
        setq( temp37, thin_old ) ;

   setq( thin_old, temp37 ) ;
```

```
      setq( this_old, car( rem ) ) ;
      setq( rem, cdr( rem ) )
      end ;
   setq( out_trains, thin_old ) ;
   setq( out_trains, decide( out_trains ) ) ;
   { Note OUT_TRAINS is updated
     explicitly here, rather than in GROUPS. }
   setq( out, out_trains ) ;

   output_aged := out ;
   deref( pulse ) ;
   deref( temp37 ) ;
   deref( rem_trains ) ;
   deref( this_train ) ;
   deref( thin_old ) ;
   deref( this_old ) ;
   deref( rem ) ;
   deref( this ) ;
   temp_ref( out )
   end ;   { output_aged }

begin   { groups }
temp66 := nil_ ;
dummy := nil_ ;
temp67 := nil_ ;
pulse := nil_ ;
out_trains := nil_ ;

setq( pulse, get_pulse ) ;

if out_trains <> nil_ then
   setq( temp67, out_trains )
else if pulse <> nil_ then
   setq( temp67, nil_ )
else
   setq( temp67, t ) ;

while temp67 = nil_ do
   begin
   setq( trains, update( pulse, trains ) ) ;
   setq( dummy, output_aged( pulse ) ) ;
   { Note OUT_TRAINS is updated in OUTPUT_AGED. }
   setq( pulse, get_pulse ) ;

   if out_trains <> nil_ then
      setq( temp67, out_trains )
   else if pulse <> nil_ then
      setq( temp67, nil_ )
   else
      setq( temp67, t )

   end ;

if out_trains <> nil_ then
   setq( dummy, put_pulse( pulse ) )
else if trains <> nil_ then
```

49

```
        begin
        setq( out_trains, decide( trains ) ) ;
        setq( trains, nil_ )
        end ;

    setq( temp66, out_trains ) ;

    groups := temp66 ;
    deref( dummy ) ;
    deref( temp67 ) ;
    deref( pulse ) ;
    deref( out_trains ) ;
    temp_ref( temp66 )
    end ;   { groups }

function group : s_exp ;
    var temp68, grp : s_exp ;

    begin   { group }
    temp68 := nil_ ;
    grp := nil_ ;

    if old_groups <> nil_ then
         begin
         setq( grp, car( old_groups ) ) ;
         setq( old_groups, cdr( old_groups ) ) ;
         setq( temp68, grp )
         end
      else
         begin
         setq( old_groups, groups ) ;

         if old_groups <> nil_ then
              setq( temp68, group )
            else
                setq( temp68, nil_ )

         end ;

    group := temp68 ;
    deref( grp ) ;
    temp_ref( temp68 )
    end ;   { group }

begin   { sort }
ref( pulses ) ;
out := nil_ ;
this_group := nil_ ;
trains := nil_ ;
old_groups := nil_ ;
last_pulse := nil_ ;
pulse_mask := nil_ ;

used_pulses := 0 ;

reset( input ) ;
```

50

```
   if eof then
        max_used_pulses := default_max_pulses
      else
         begin
         readln( max_used_pulses ) ;
         readln
         end ;

   setq( pulse_mask, printf( get_mask ) ) ;

   setq( this_group, group ) ;
   repeat
      summarize( this_group, group_summary ) ;
      write( summarys, group_summary ) ;
      setq( this_group, group )
   until this_group = nil_ ;

   setq( out, nil_ ) ;

   sort := out ;
   deref( pulses ) ;
   deref( this_group ) ;
   deref( trains ) ;
   deref( old_groups ) ;
   deref( last_pulse ) ;
   deref( pulse_mask ) ;
   temp_.ef( out )
   end ;  { sort }

function atom_with_time( time : microseconds ) : s_exp ;
   var out : s_exp ;

   begin  { atom_with_time }
   out := nil_ ;

   setq( out, get_atom ) ;

   out ^. toa_  := time ;
   out ^. bearing := 0.0 ;
   out ^. amplitude := 0.0 ;

   atom_with_time := out ;
   temp_ref( out )
   end ;  { atom_with_time }

function train( start, pri : microseconds ; n : integer ) : s_exp ;
   var out : s_exp ;

   begin  { train }
   out := nil_ ;

   if n = 0 then
        setq( out, nil_ )
      else
        setq( out, cons( atom_with_time( start ),
                         train( start + pri, pri, n - 1 ) ) ) ;
```

51

```
    train := out ;
    temp_ref( out )
    end ;   { train }

function merge2( list1, list2 : s_exp ) : s_exp ;
    var temp12, temp11, out : s_exp ;

    begin   { merge2 }
    ref( list1 ) ;
    ref( list2 ) ;
    out := nil_ ;
    temp12 := nil_ ;
    temp11 := nil_ ;

    if list2 <> nil_ then
        setq( temp11, nil_ )
      else
        setq( temp11, t ) ;

    if temp11 <> nil_ then
        setq( out, list1 )
      else
        begin

        if list1 <> nil_ then
            setq( temp12, nil_ )
          else
            setq( temp12, t ) ;

        if temp12 <> nil_ then
          setq( out, list2 )
        else if toa( car( list1 ) ) < toa( car( list2 ) ) then
          setq( out, cons( car( list1 ),
                              merge2( cdr( list1 ), list2 ) ) )
        else
          setq( out, cons( car( list2 ),
                              merge2( list1, cdr( list2 ) ) ) )
        end ;

    merge2 := out ;
    deref( list1 ) ;
    deref( list2 ) ;
    deref( temp12 ) ;
    deref( temp11 ) ;
    temp_ref( out )
    end ;   { merge2 }

  { 1111111111  lisp system initialization:  1111111111 }

begin  { main program }
new( nil_, false ) ;
nil_ ^. atomic := true ;
nil_ ^. ref_count := 1 ;
nil_ ^. car_ := nil_ ;
nil_ ^. cdr_ := nil_ ;
```

```
garbage.lists := nil_ ;
garbage.atomic := nil_ ;

conses_created := 0 ;
atoms_created := 0 ;

new( t, true ) ;
t ^. atomic := true ;
t ^. ref_count := 1 ;

   { uuuuuuuuuu user main program here:  uuuuuuuuuu }

dummy := nil_ ;
from_file := nil_ ;

setq( from_file, get_atom ) ;    { used as a constant }

setq( dummy, sort( from_file ) ) ;

deref( dummy ) ;
deref( from_file ) ;

 { llllllllll  lisp debugging info:  llllllllll }

writeln ;
writeln ;

writeln('  conses created:', conses_created ) ;

writeln ;

writeln('  atoms created:', atoms_created ) ;

writeln ;

writeln('  conses in garbage:', stack_length( garbage.lists ) ) ;

writeln ;

writeln('  atoms in garbage:', length( garbage.atomic ) )

end .  { main program }
```

RATFOR Input Routines for PASCAL Program

```
# global macro definitions

[   # add assert statement
# version jc00

# syntax: <assert statement> ::=    @assert ( <condition> ) @istrue
#                                  | @assert ( <condition> ) @isfalse

# if any assertions are present in a routine then either
# @assertions or @noassertions must be expanded prior to
# the first assertion

define( @assertions,
   define( @assert, if )
   define( @istrue, ; else [ write(6,12345) ; call abend ] )
   define( @isfalse, [ write(6,12345) ; call abend ] )
   12345 format('1error in assertion')
   )

define( @noassertions,
   define( @assert, # )
   define( @istrue,   )
   define( @isfalse,  )
   )
]


[
define(@radar, 2)
define(@df,    1)
define(@ok,  0 )                        # see asc subprogram library..
define(@eof, (8*16**3 + 16**2)) # ..notebook page i9.000 re: word6
define(@bor, -1)                        # beginning of record
]
end


# unpack      - get an unformatted radar or df detection -
  subroutine unpack( status, type, data )
  @assertions  # turn-on

  [   # arguments    ( #i = input, #o = output )
   integer*4 status,   #o @ok or @eof
            type,      #o either @radar or @df
            data(4)    #o unpacked detection
            #   element : type = @df       @radar
            #   --------:----------------------
            #     1    :          sec       msec32
            #     2    :          usec      range
            #     3    :          east      az
            #     4    :          south     el
  ]

c ***********************************************************
c
```

```
c version jc08.1
c started          4/15/80
c last modified    2/25/81
c by j. coleman, nrl code 5312, wash dc 20375, 767-2399
c
c unpacks and returns a detection from an experimental
c data file.  the experimental quantities are left in
c their original units.  acnm = data.
c
c differs from version jc08 in that ^= has been used for .ne.
c ***********************************************************

  [  # non-common variables & constants
  integer*4 word16,    # function to fetch next word
            word,      # word fetched from file (16 bit right just)
            wrdnr,     # word number within detection (not in order)
            wrdtyp,    # word type indicator (1 => radar, 2 => df)
            i,j,       # loop counters
            msknr,     # mask for word number field
            msktyp,    # mask for word type field
            msk2(4),   # masks for fields contributing to data(2)
            shft(4)    # shifts for fields contributing to data(2)

  data msknr  / z00000003 /,     # bits 0, 1
       msktyp / z00008000 /,     # bit 15
       msk2   / z0000001c ,      # bits 2-4
                z0000001c ,      # bits 2-4
                z00007ffc ,      # bits 2-14
                z00000004 /,     # bit 2
       shft   / -2,
                 1,
                 4,
                17 /
  ]


  [  # common /unpbuf/
  integer*4 buffer(4,2)    # buffer to accumulate detections
  common /unpbuf/ buffer
  ]


  repeat     # until complete detection test is passed
     repeat     # until last word of a detection has been buffered
        [
        status = word16('data     ',status,word)
        if( status == @eof )
           return
        if( status == @bor )  # beginning of record
           [
           do i = 1, 2
              do j = 1, 4
                 buffer(j,i) = 0
           status = @ok
           ]
        @assert( status == @ok )@istrue
        @assert( word < 2**16  )@istrue  # zeros in left halfword
```

```
        wrdnr = 1 + and(word, msknr)       # wrdnr in [1..4]
        wrdtyp = 1 + lshf(and(word, msktyp), -15) # wrdtyp in [1,2]
        buffer(wrdnr, wrdtyp) = word
        ]
    until( wrdnr == 1 )  # last word of a detection
 until(   buffer(2,wrdtyp) ^= 0
      & buffer(3,wrdtyp) ^= 0
      & buffer(4,wrdtyp) ^= 0 )  # test for complete detection

 if( wrdtyp == 2 )
        [
        type = @radar
        data(1) = lshf( and( buffer(2,2), compl( msktyp ) ), -2 )
        data(2) = lshf( and( buffer(3,2), compl( msktyp ) ), -3 )
        data(3) = lshf( and( buffer(4,2), compl( msktyp ) ), -3 )
        data(4) = lshf( and( buffer(1,2), compl( msktyp ) ), -2 )
        ]
    else
        [
        @assert( wrdtyp == 1 )@istrue
        type = @df
        data(1) = lshf( buffer(4,1), -3 )   # sec
        data(2) = 0
        do i = 1, 4      # usec
           data(2) = or(data(2),lshf(and(msk2(i),buffer(i,1)),shft(i)))
        data(3) = lshf( buffer(2,1), -5 )    # east
        data(4) = lshf( buffer(1,1), -5 )    # south
        ]

 do i = 1, 4   # immunizes for a class of data recording errors
    buffer(i, wrdtyp) = 0

 return
 end


# word16         - get a 16 bit word from a data file -
 integer function word16( acnm, stat, word )  # word16 = stat

 [  # arguments    ( #i = input, #o = output )
 integer*4 acnm(2),  #i 8 char file acnm, left just, blank padded
           stat,     #o a status, in [ @ok, @bor, @eof ]
           word      #o 16 bit word from file, right just, 0 padded
 ]

c ************************************************************
c version jc01.1
c started        4/16/80
c last modified  2/25/81
c by j. coleman, code 5312 nrl, wash dc 20375, 767-2399
c
c returns successive 16 bit words from the file <acnm> on
c successive calls.  the file may have multiple records
c
c differs from version jc01 in that ^= has been used for .ne.
c ************************************************************
```

```
[  # common /w16buf/  'static', local, *init in block data w16bd
define(   @inpbufsize, 16384   )
integer*4 inpbuf(@inpbufsize),   # input buffer for readfl
          last16,                #* index of last used inbuf halfword
          mskwrd,                #* mask for right halfword= z0000ffff
          param(14)              #* readfl interface

equivalence ( param(1), fname ),
            ( param(4), reclen ),
            ( param(6), status )

integer*4 fname(2),              #* file to read from (acnm)
          reclen,                #* record length (bytes) read
          status                 #* status returned by readfl

common /w16buf/ last16, mskwrd, param, inpbuf
]


stat = @ok  # default, subject to change

if( last16 >= reclen/2 )  # ignore any odd nred byte at record end
    [
    last16 = 0

    fname(1) = acnm(1) ; fname(2) = acnm(2)
    call readfl( param, inpbuf )

    if( (reclen+3)/4 > @inpbufsize )  # overflow?
       [
       write(6,2) reclen ; 2 format('1reclen =',i7)
       call abend
       ]
    if( status == @eof )
       [
       stat = @eof
       word16 = @eof
       return
       ]
    if( status ^= @ok )
       [
       write(6,1) status ; 1 format('1status = ',z8,' hex')
       call abend
       ]
    stat = @bor   # beginning of record
    ]
if( mod( last16, 2 ) == 0 )
    word = lshf( inpbuf(1+last16/2), -16 )    # integer divide
    else
    word = and( inpbuf(1+last16/2), mskwrd )  # integer divide

last16 = last16 + 1

word16 = stat

return
```

57

```
    end


#  w16bd    - block data for word16 -
  block data

c vers jc02

  [  # common /w16buf/  'static', local, *init in block data w16bd
  define(   @inpbufsize, 16384   )
  integer*4 inpbuf(@inpbufsize),  # input buffer for readfl
           last16,                #* index of last used inbuf halfword
           mskwrd,                #* mask for right halfword= z0000ffff
           param(14)              #* readfl interface

  equivalence ( param(1), fname  ),
             ( param(4), reclen ),
             ( param(6), status )

  integer*4 fname(2),             #* file to read from (acnm)
           reclen,                #* record length (bytes) read
           status                 #* status returned by readfl

  common /w16buf/ last16, mskwrd, param, inpbuf
  ]

  data last16    /  @inpbufsize  /,
      mskwrd     /  z0000ffff    /,
      reclen     /  0            /,  # must have last16 >= reclen/2
      param(3) /   0            /,
      param(7) /   0            /,
      param(9) /   0            /,
      param(13)/   0            /,
      param(14)/   0            /
  end
```

DATE
FILMED

8